华章程序员书库

[PACKT] PUBLISHING

HZ BOOKS 华章IT

C#多线程编程实战

Multithreading in C# 5.0 Cookbook



[美] Eugene Agafonov 著

黄博文 黄辉兰 译

机械工业出版社
China Machine Press

Multithreading in C# 5.0 Cookbook

在计算机处理器发展为包含越来越多的核心的时期，多线程是创建可伸缩性、高效的、高响应性应用程序的关键因素。如果你没有正确地使用多线程，它会导致难以捉摸的问题，需要你花费大量时间去解决。因此，现代应用程序开发人员必须深入理解多线程。

本书是一本通俗易懂的C#多线程编程指南，通过70多个容易理解的示例，循序渐进地讲解C#5.0中的异步及并发编程，引导读者了解Windows下C#多线程编程的多样性。

通过阅读本书，你将学到：

- 使用原始线程、异步线程，以及协调它们工作
- 使用任务并行库设计自己的异步API
- 使用C# 5.0异步语言特性
- 使用I/O线程提高服务器应用程序性能
- 使用PLINQ并行化LINQ查询
- 使用Windows 8异步API
- 使用Reactive Extensions运行异步操作并管理选项

[PACKT]
PUBLISHING

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/程序设计

ISBN 978-7-111-49348-8

9 787111 493488 >

定价：49.00元

华章程序员书库



Multithreading in C# 5.0 Cookbook

C#多线程编程实战

[美] Eugene Agafonov 著

黄博文 黄辉兰 译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

C# 多线程编程实战 /（美）阿格佛温（Agafonov，E.）著；黄博文等译 . —北京：机械工业出版社，2015.3
（华章程序员书库）
书名原文：Multithreading in C# 5.0 Cookbook

ISBN 978-7-111-49348-8

I. C… II. ①阿… ②黄… ③黄… III. C 语言 – 程序设计 IV. TP312

中国版本图书馆 CIP 数据核字（2015）第 029657 号

本书版权登记号：图字：01-2014-2038

Multithreading in C# 5.0 Cookbook (ISBN: 978-1-84969-764-4).

Copyright © 2013 Packt Publishing. First published in the English language under the title "Multithreading in C# 5.0 Cookbook".

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2015 by China Machine Press.

本书中文简体字版由 Packt Publishing 授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

C# 多线程编程实战

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）
责任编辑：陈佳媛 责任校对：董纪丽
印 刷：三河市宏图印务有限公司 版 次：2015 年 3 月第 1 版第 1 次印刷
开 本：186mm × 240mm 1/16 印 张：13.5
书 号：ISBN 978-7-111-49348-8 定 价：49.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

版权所有 • 侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

Preface 前　言

不久前，典型的个人计算机的CPU还只有一个计算核心，并且功耗足以煎熟鸡蛋。2005年，英特尔推出了其首款多核心CPU，从此计算机开始向不同的方向发展。低耗电量及多个计算核心变得比提高行计算（row computing）核心性能更重要。这也导致了编程范式的改变。现在我们需要学习如何有效地使用所有CPU核心来最优化性能，并同时通过在特定时间只运行需要的程序来节省电池电量。除此之外，我们在编写服务器端应用程序时需要有效地利用多个CPU核心，甚至多台计算机来支持尽可能多的用户。

为了创建这样的应用程序，你需要学习如何在程序中有效地使用多个CPU核心。如果你使用的是Microsoft .NET开发平台以及C#编程语言，那么本书将是一个编写高性能、高响应性的应用程序的完美起点。

本书的目的是给你提供C#中多线程以及并行编程的详尽指导。我们将从基本概念开始，每章主题比前一章都有所拔高，最后展示现实世界中的并行编程模式以及Windows商店应用示例。

本书内容

第1章介绍了C#中基本的线程操作。本章解释了什么是线程，使用线程的优缺点，以及与线程相关的其他重要方面。

第2章描述了线程交互细节。你将了解为何我们需要协调线程，以及协调组织线程的不同方式。

第3章解释了线程池概念。本章展示了如何使用线程池，如何执行异步操作，以及使用线程池的好的和不好的实践。

第4章深入讲解了任务并行库（Task Parallel Library，TPL）框架。本章讲述了TPL的所有重要方面，包括任务组合、异常管理及取消操作等。

第 5 章深入解释了 C# 5.0 中的新功能——异步方法。你将了解 async 和 await 关键字，如何在不同的场景中使用它们，以及 await 底层工作机制。

第 6 章描述了 .NET 框架中并行算法的标准数据结构，并为每种数据结构展示了示例编程场景。

第 7 章深入讲解了并行 LINQ 基础设施。本章讲述了任务和数据并行度，并行化 LINQ 查询，调整并行选项，分割查询，集合并行查询结果等内容。

第 8 章解释了如何以及何时使用 Reactive Extensions 框架。你将学习如何组合事件，如何对事件序列执行 LINQ 查询等。

第 9 章深入讲解了异步 I/O 进程，包括文件、网络及数据库等场景。

第 10 章列出了针对常见的并行编程问题的解决方案。

第 11 章覆盖了基于 Windows 8 的编写异步应用程序的场景。你将学习如何使用 Windows 8 异步 API，以及如何在 Windows 商店应用中完成后台工作。

准备事项

我们需要 Windows 桌面版本的 Microsoft Visual Studio Express 2012 来完成大多数章节的学习。第 11 章中的小节还需要 Windows 8 操作系统和 Visual Studio Express 2012 编译 Windows 商店应用。

读者对象

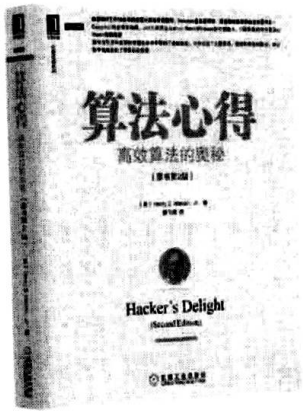
本书的读者对象为没有或只有少量多线程及异步和并发编程背景的 C# 开发人员。本书涵盖了 C# 和 .NET 生态系统中从基础概念到复杂编程模式及算法的很多与多线程相关的主题。

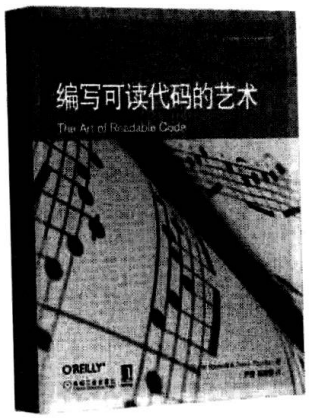
下载示例代码

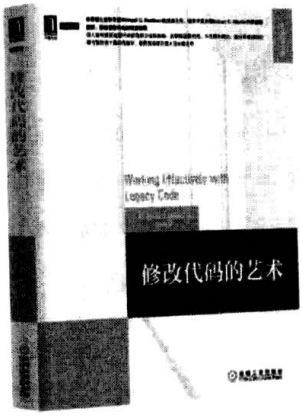
本书的示例代码可登录华章网站（http://www.hzbook.com）中的本书页面下载。

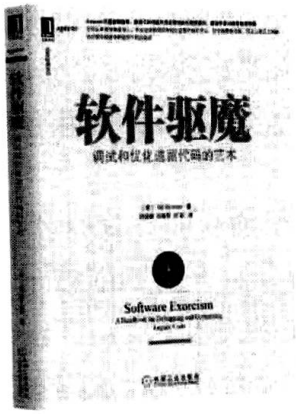
推荐阅读


算法精解
C语言描述
O'REILLY


算法心得
高效算法的奥秘
Hacker's Delight


编写可读代码的艺术
O'REILLY


修改代码的艺术

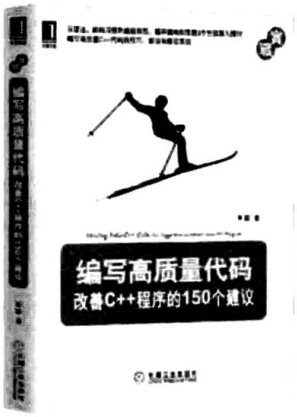

软件驱魔
调试和优化遗留代码的艺术
Software Exorcism


代码之殇

推荐阅读
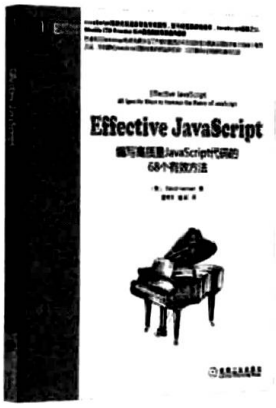

编写高质量代码
改善Java程序的151个建议
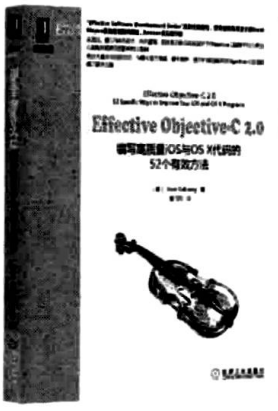

编写高质量代码
改善C#程序的157个建议
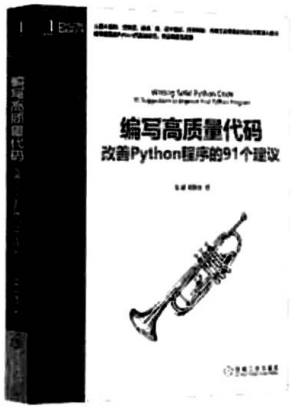

编写高质量代码
改善C++程序的150个建议


编写高质量代码
Web前端开发修炼之道


编写高质量代码
改善JavaScript程序的188个建议

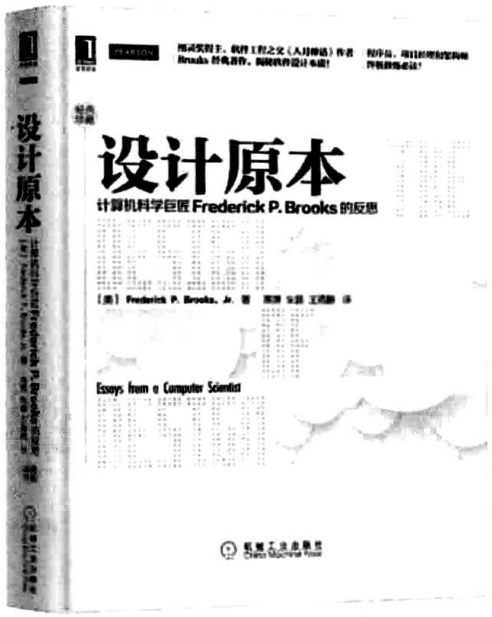

Effective JavaScript
编写高质量JavaScript代码的
68个有效方法


Effective Objective-C 2.0
编写高质量iOS与OS X代码的
52个有效方法


编写高质量代码
改善Python程序的91个建议

推荐阅读





人件（原书第3版）

作者：（美）Tom DeMarco 等 ISBN：978-7-111-47436-4 定价：69.00元

公认对软件行业影响最大、最具价值的著作之一，历时15年全面更新
与《人月神话》共同被誉为软件图书领域最为璀璨的“双子星”，近30年全球畅销不衰

在软件管理领域，很少有著作能够与本书媲美。全书从管理人力资源、创建健康的办公环境、雇用并留用正确的人、高效团队形成、改造企业文化和快乐工作等多个角度阐释了如何思考和管理软件开发的最大问题——人（而不是技术），以得到高效的项目和团队。

设计原本——计算机科学巨匠Frederick P. Brooks的反思（经典珍藏）

作者：（美）Frederick P. Brooks,Jr. ISBN：978-7-111-41626-5 定价：79.00元

图灵奖得主、《人月神话》作者Brooks封笔之作，揭秘软件设计神话！
程序员、项目经理和架构师必读的一本书！

《设计原本》开启了软件工程全新的“后理性时代”，完成了从破到立的圆满循环，具有划时代的重大里程碑意义，是每位从事软件行业的程序员、项目经理和架构师都应该反复研读的经典著作。全书以设计理念为核心，从对设计模型的探讨入手，讨论了有关设计的若干重大问题：设计过程的建立、设计协作的规划、设计范本的固化、设计演化的管控，以及设计师的发现和培养。

Contents 目　录

前　言

第1章　线程基础 …… 1

1.1　简介 …… 1

1.2　使用 C# 创建线程 …… 2

1.2.1　准备工作 …… 2

1.2.2　实现方式 …… 2

1.2.3　工作原理 …… 4

1.3　暂停线程 …… 4

1.3.1　准备工作 …… 5

1.3.2　实现方式 …… 5

1.3.3　工作原理 …… 5

1.4　线程等待 …… 6

1.4.1　准备工作 …… 6

1.4.2　实现方式 …… 6

1.4.3　工作原理 …… 7

1.5　终止线程 …… 7

1.5.1　准备工作 …… 7

1.5.2　实现方式 …… 7

1.5.3　工作原理 …… 8

1.6　检测线程状态 …… 8

1.6.1　准备工作 …… 8

1.6.2 实现方式 …… 8
1.6.3 工作原理 …… 9
1.7 线程优先级 …… 10
1.7.1 准备工作 …… 10
1.7.2 实现方式 …… 10
1.7.3 工作原理 …… 11
1.8 前台线程和后台线程 …… 12
1.8.1 准备工作 …… 12
1.8.2 实现方式 …… 12
1.8.3 工作原理 …… 13
1.9 向线程传递参数 …… 13
1.9.1 准备工作 …… 13
1.9.2 实现方式 …… 13
1.9.3 工作原理 …… 15
1.9.4 更多信息 …… 15
1.10 使用 C# 中的 lock 关键字 …… 16
1.10.1 准备工作 …… 16
1.10.2 实现方式 …… 16
1.10.3 工作原理 …… 18
1.11 使用 Monitor 类锁定资源 …… 18
1.11.1 准备工作 …… 18
1.11.2 实现方式 …… 18
1.11.3 工作原理 …… 20
1.12 处理异常 …… 20
1.12.1 准备工作 …… 20
1.12.2 实现方式 …… 21
1.12.3 工作原理 …… 22

第 2 章 线程同步 …… 23

2.1 简介 …… 23
2.2 执行基本的原子操作 …… 24

2.2.1 准备工作 ····· 24
2.2.2 实现方式 ····· 24
2.2.3 工作原理 ····· 26
2.3 使用 Mutex 类 ····· 27
2.3.1 准备工作 ····· 27
2.3.2 实现方式 ····· 27
2.3.3 工作原理 ····· 27
2.4 使用 SemaphoreSlim 类 ····· 28
2.4.1 准备工作 ····· 28
2.4.2 实现方式 ····· 28
2.4.3 工作原理 ····· 29
2.4.4 更多信息 ····· 29
2.5 使用 AutoResetEvent 类 ····· 29
2.5.1 准备工作 ····· 30
2.5.2 实现方式 ····· 30
2.5.3 工作原理 ····· 31
2.6 使用 ManualResetEventSlim 类 ····· 31
2.6.1 准备工作 ····· 31
2.6.2 实现方式 ····· 31
2.6.3 工作原理 ····· 32
2.6.4 更多信息 ····· 33
2.7 使用 CountDownEvent 类 ····· 33
2.7.1 准备工作 ····· 33
2.7.2 实现方式 ····· 33
2.7.3 工作原理 ····· 34
2.8 使用 Barrier 类 ····· 34
2.8.1 准备工作 ····· 34
2.8.2 实现方式 ····· 34
2.8.3 工作原理 ····· 35
2.9 使用 ReaderWriterLockSlim 类 ····· 35
2.9.1 准备工作 ····· 36

2.9.2 实现方式 …… 36
2.9.3 工作原理 …… 37
2.10 使用 SpinWait 类 …… 38
2.10.1 准备工作 …… 38
2.10.2 实现方式 …… 38
2.10.3 工作原理 …… 39

第 3 章 使用线程池 …… 40

3.1 简介 …… 40
3.2 在线程池中调用委托 …… 41
3.2.1 准备工作 …… 42
3.2.2 实现方式 …… 42
3.2.3 工作原理 …… 43
3.3 向线程池中放入异步操作 …… 44
3.3.1 准备工作 …… 44
3.3.2 实现方式 …… 44
3.3.3 工作原理 …… 45
3.4 线程池与并行度 …… 45
3.4.1 准备工作 …… 46
3.4.2 实现方式 …… 46
3.4.3 工作原理 …… 47
3.5 实现一个取消选项 …… 47
3.5.1 准备工作 …… 47
3.5.2 实现方式 …… 48
3.5.3 工作原理 …… 49
3.6 在线程池中使用等待事件处理器及超时 …… 50
3.6.1 准备工作 …… 50
3.6.2 实现方式 …… 50
3.6.3 工作原理 …… 51
3.6.4 更多信息 …… 51
3.7 使用计时器 …… 52

3.7.1 准备工作 …… 52
3.7.2 实现方式 …… 52
3.7.3 工作原理 …… 53
3.8 使用 BackgroundWorker 组件 …… 53
3.8.1 准备工作 …… 53
3.8.2 实现方式 …… 53
3.8.3 工作原理 …… 55

第 4 章 使用任务并行库 …… 56

4.1 简介 …… 56
4.2 创建任务 …… 57
4.2.1 准备工作 …… 58
4.2.2 实现方式 …… 58
4.2.3 工作原理 …… 59
4.3 使用任务执行基本的操作 …… 59
4.3.1 准备工作 …… 60
4.3.2 实现方式 …… 60
4.3.3 工作原理 …… 61
4.4 组合任务 …… 61
4.4.1 准备工作 …… 61
4.4.2 实现方式 …… 61
4.4.3 工作原理 …… 63
4.5 将 APM 模式转换为任务 …… 63
4.5.1 准备工作 …… 64
4.5.2 实现方式 …… 64
4.5.3 工作原理 …… 66
4.6 将 EAP 模式转换为任务 …… 66
4.6.1 准备工作 …… 66
4.6.2 实现方式 …… 67
4.6.3 工作原理 …… 68
4.7 实现取消选项 …… 68

4.7.1 准备工作 …… 68
4.7.2 实现方式 …… 68
4.7.3 工作原理 …… 69
4.8 处理任务中的异常 …… 69
4.8.1 准备工作 …… 70
4.8.2 实现方式 …… 70
4.8.3 工作原理 …… 71
4.8.4 更多信息 …… 71
4.9 并行运行任务 …… 72
4.9.1 准备工作 …… 72
4.9.2 实现方式 …… 72
4.9.3 工作原理 …… 73
4.10 使用 TaskScheduler 配置任务的执行 …… 73
4.10.1 准备工作 …… 73
4.10.2 实现方式 …… 74
4.10.3 工作原理 …… 76

第 5 章 使用 C# 5.0 …… 78

5.1 简介 …… 78
5.2 使用 await 操作符获取异步任务结果 …… 80
5.2.1 准备工作 …… 80
5.2.2 实现方式 …… 80
5.2.3 工作原理 …… 81
5.3 在 lambda 表达式中使用 await 操作符 …… 82
5.3.1 准备工作 …… 82
5.3.2 实现方式 …… 82
5.3.3 工作原理 …… 83
5.4 对连续的异步任务使用 await 操作符 …… 83
5.4.1 准备工作 …… 84
5.4.2 实现方式 …… 84
5.4.3 工作原理 …… 85

5.5 对并行执行的异步任务使用 await 操作符 …… 86
5.5.1 准备工作 …… 86
5.5.2 实现方式 …… 86
5.5.3 工作原理 …… 87
5.6 处理异步操作中的异常 …… 88
5.6.1 准备工作 …… 88
5.6.2 实现方式 …… 88
5.6.3 工作原理 …… 89
5.7 避免使用捕获的同步上下文 …… 90
5.7.1 准备工作 …… 90
5.7.2 实现方式 …… 90
5.7.3 工作原理 …… 92
5.8 使用 async void 方法 …… 93
5.8.1 准备工作 …… 93
5.8.2 实现方式 …… 93
5.8.3 工作原理 …… 95
5.9 设计一个自定义的 awaitable 类型 …… 96
5.9.1 准备工作 …… 96
5.9.2 实现方式 …… 96
5.9.3 工作原理 …… 97
5.10 对动态类型使用 await …… 98
5.10.1 准备工作 …… 99
5.10.2 实现方式 …… 99
5.10.3 工作原理 …… 101

第 6 章 使用并发集合 …… 102

6.1 简介 …… 102
6.2 使用 ConcurrentDictionary …… 103
6.2.1 准备工作 …… 104
6.2.2 实现方式 …… 104
6.2.3 工作原理 …… 105

6.3 使用 ConcurrentQueue 实现异步处理 …… 105
6.3.1 准备工作 …… 106
6.3.2 实现方式 …… 106
6.3.3 工作原理 …… 107
6.4 改变 ConcurrentStack 异步处理顺序 …… 108
6.4.1 准备工作 …… 108
6.4.2 实现方式 …… 108
6.4.3 工作原理 …… 109
6.5 使用 ConcurrentBag 创建一个可扩展的爬虫 …… 110
6.5.1 准备工作 …… 110
6.5.2 实现方式 …… 110
6.5.3 工作原理 …… 113
6.6 使用 BlockingCollection 进行异步处理 …… 113
6.6.1 准备工作 …… 113
6.6.2 实现方式 …… 113
6.6.3 工作原理 …… 115

第 7 章 使用 PLINQ …… 116

7.1 简介 …… 116
7.2 使用 Parallel 类 …… 117
7.2.1 准备工作 …… 118
7.2.2 实现方式 …… 118
7.2.3 工作原理 …… 119
7.3 并行化 LINQ 查询 …… 119
7.3.1 准备工作 …… 119
7.3.2 实现方式 …… 120
7.3.3 工作原理 …… 122
7.4 调整 PLINQ 查询的参数 …… 122
7.4.1 准备工作 …… 122
7.4.2 实现方式 …… 123
7.4.3 工作原理 …… 124

7.5 处理 PLINQ 查询中的异常 …… 125
7.5.1 准备工作 …… 125
7.5.2 实现方式 …… 125
7.5.3 工作原理 …… 126
7.6 管理 PLINQ 查询中的数据分区 …… 126
7.6.1 准备工作 …… 127
7.6.2 实现方式 …… 127
7.6.3 工作原理 …… 128
7.7 为 PLINQ 查询创建一个自定义的聚合器 …… 129
7.7.1 准备工作 …… 129
7.7.2 实现方式 …… 129
7.7.3 工作原理 …… 131

第 8 章 使用 Reactive Extensions …… 132

8.1 简介 …… 132
8.2 将普通集合转换为异步的可观察集合 …… 133
8.2.1 准备工作 …… 133
8.2.2 实现方式 …… 133
8.2.3 工作原理 …… 135
8.3 编写自定义的可观察对象 …… 136
8.3.1 准备工作 …… 136
8.3.2 实现方式 …… 136
8.3.3 工作原理 …… 138
8.4 使用 Subject …… 138
8.4.1 准备工作 …… 138
8.4.2 实现方式 …… 138
8.4.3 工作原理 …… 140
8.5 创建可观察的对象 …… 141
8.5.1 准备工作 …… 141
8.5.2 实现方式 …… 141
8.5.3 工作原理 …… 143

8.6 对可观察的集合使用 LINQ 查询 …… 144
8.6.1 准备工作 …… 144
8.6.2 实现方式 …… 144
8.6.3 工作原理 …… 145
8.7 使用 Rx 创建异步操作 …… 146
8.7.1 准备工作 …… 146
8.7.2 实现方式 …… 146
8.7.3 工作原理 …… 148

第 9 章 使用异步 I/O …… 150

9.1 简介 …… 150
9.2 异步地使用文件 …… 152
9.2.1 准备工作 …… 152
9.2.2 实现方式 …… 152
9.2.3 工作原理 …… 154
9.3 编写一个异步的 HTTP 服务器和客户端 …… 155
9.3.1 准备工作 …… 155
9.3.2 实现方式 …… 155
9.3.3 工作原理 …… 157
9.4 异步操作数据库 …… 157
9.4.1 准备工作 …… 157
9.4.2 实现方式 …… 158
9.4.3 工作原理 …… 160
9.5 异步调用 WCF 服务 …… 161
9.5.1 准备工作 …… 161
9.5.2 实现方式 …… 161
9.5.3 工作原理 …… 164

第 10 章 并行编程模式 …… 165

10.1 简介 …… 165
10.2 实现惰性求值的共享状态 …… 166

10.2.1 准备工作 …… 166
10.2.2 实现方式 …… 166
10.2.3 工作原理 …… 169
10.3 使用 BlockingCollection 实现并行管道 …… 170
10.3.1 准备工作 …… 170
10.3.2 实现方式 …… 170
10.3.3 工作原理 …… 174
10.4 使用 TPL 数据流实现并行管道 …… 175
10.4.1 准备工作 …… 175
10.4.2 实现方式 …… 175
10.4.3 工作原理 …… 177
10.5 使用 PLINQ 实现 Map/Reduce 模式 …… 178
10.5.1 准备工作 …… 178
10.5.2 实现方式 …… 178
10.5.3 工作原理 …… 181

第 11 章 更多信息 …… 183

11.1 简介 …… 183
11.2 在 Windows 商店应用中使用计时器 …… 184
11.2.1 准备工作 …… 184
11.2.2 实现方式 …… 184
11.2.3 工作原理 …… 188
11.3 在通常的应用程序中使用 WinRT …… 189
11.3.1 准备工作 …… 189
11.3.2 实现方式 …… 189
11.3.3 工作原理 …… 191
11.4 在 Windows 商店应用中使用 BackgroundTask …… 192
11.4.1 准备工作 …… 192
11.4.2 实现方式 …… 192
11.4.3 工作原理 …… 197



第 1 章 Chapter 1

线程基础

本章将涵盖 C# 中使用线程的基本操作。

你将学到以下内容：

- 使用 C# 创建线程
- 暂停线程
- 线程等待
- 终止线程
- 检测线程状态
- 线程优先级
- 前台线程和后台线程
- 向线程传参
- 使用 C# 中的 lock 关键字
- 使用 Monitor 类锁定资源
- 处理异常

1.1 简介

过去普通计算机只有一个计算单元，不能同时执行多个计算任务。然而操作系统却已经可以同时运行多个应用程序，即实现了多任务的概念。为了防止一个应用程序控制 CPU 而导致其他应用程序和操作系统本身永远被挂起这一可能情况，操作系统不得不使

用某种方式将物理计算单元分割为一些虚拟的进程，并给予每个执行程序一定量的计算能力。此外，操作系统必须始终能够优先访问 CPU，并能调整不同程序访问 CPU 的优先级。线程正是这一概念的实现。可以认为线程是一个虚拟进程，用于独立运行一个特定的程序。



请记住线程会消耗大量的操作系统资源。多个线程共享一个物理处理器将导致操作系统忙于管理这些线程，而无法运行程序。

因此，虽然有可能提高计算机的处理器能力，使得计算机每秒钟能执行越来越多的命令，但是使用线程通常是一个操作系统任务，试图在单核 CPU 上并行执行计算任务是没有意义的，因为这比顺序运行会花费更多的时间。然而，当处理器拥有多核时，过去的应用程序则不能使用这个优势，因为它们只使用了一个处理核心。

为了有效地利用现代处理器的计算能力，使用某种方式让程序能使用不止一个处理核心是非常重要的。这需要组织多个线程间的通信和相互同步。

本章中的内容将关注于使用 C# 语言执行一些非常基本的线程操作。我们将介绍线程的生命周期，其包括创建线程、挂起线程、线程等待，以及中止线程。然后我们会介绍一些基本的线程同步技术。

1.2 使用 C# 创建线程

在接下来的内容中，我们将使用 Visual Studio 2012 作为主要的工具来使用 C# 编写多线程程序。本节将展示如何创建一个新的 C# 程序，并在该程序中使用线程。



从微软官方网站可以下载免费的 Visual Studio 2012 Express 版本。我们需要 Windows 桌面版本的 Visual Studio 2012 Express 来完成大多数示例，还需要 Windows 8 版本的 Visual Studio 2012 Express 版本完成特定于 Windows 8 的示例。

1.2.1 准备工作

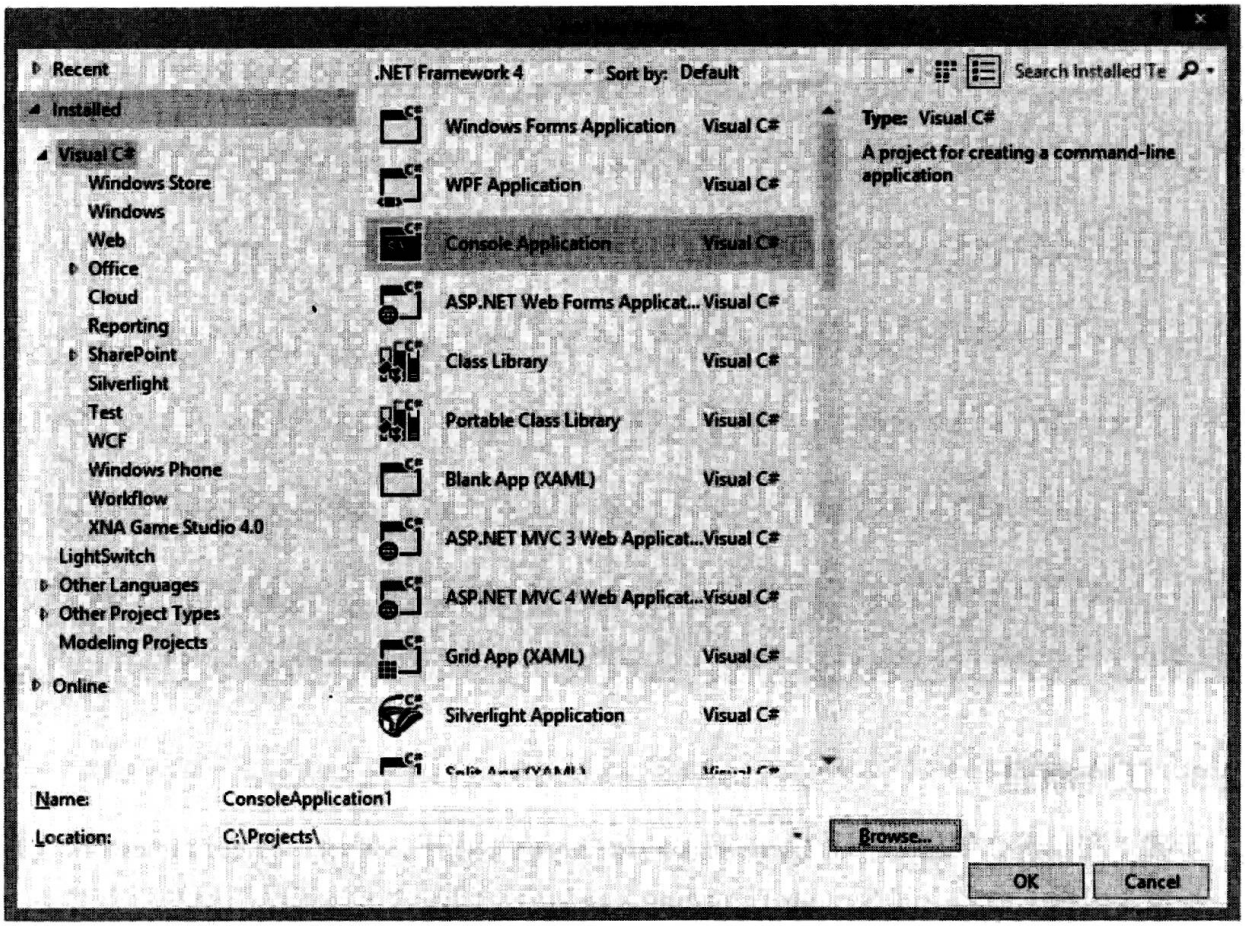
为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe1 目录中。

1.2.2 实现方式

请执行以下步骤来了解如何创建一个新的 C# 程序，并在其中使用线程：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序工程。

2. 确保该工程使用 .NET Framework4.0 或以上版本。



3. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

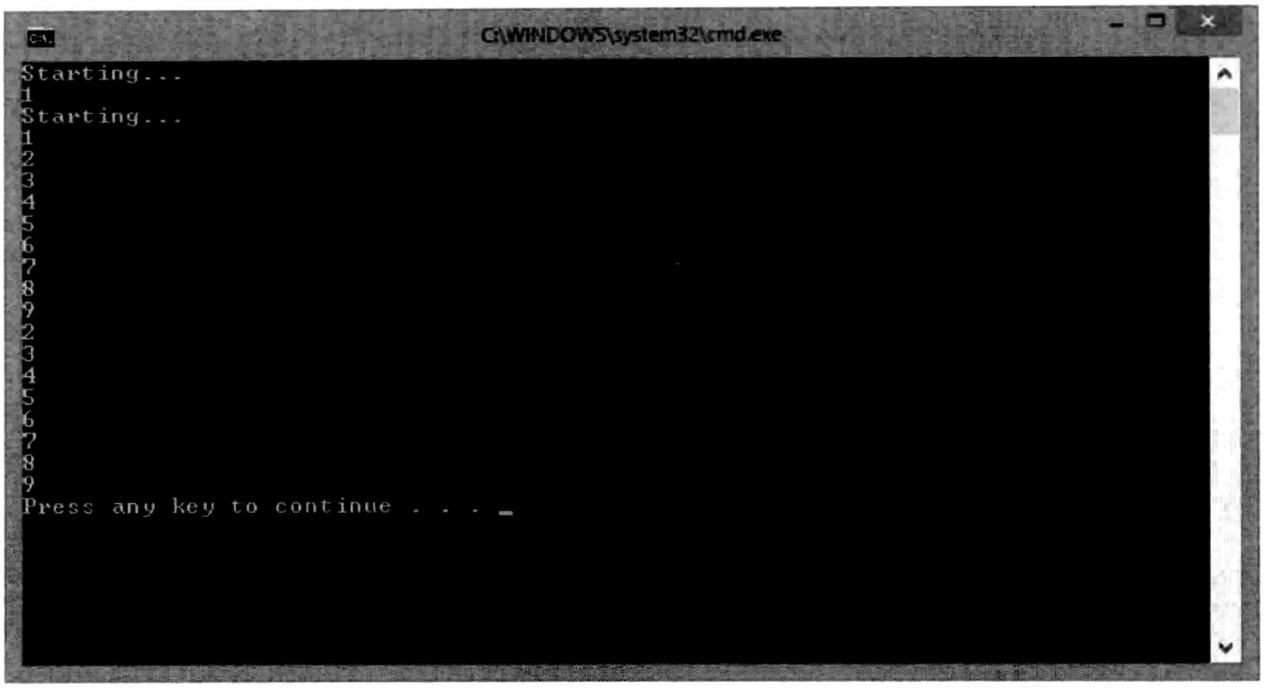
4. 在 Main 方法下面加入以下代码片段：

```
static void PrintNumbers()
{
  Console.WriteLine("Starting...");
  for (int i = 1; i < 10; i++)
  {
    Console.WriteLine(i);
  }
}
```

5. 在 Main 方法中加入以下代码片段：

```
Thread t = new Thread(PrintNumbers);
t.Start();
PrintNumbers();
```

6. 运行程序。输出如下所示：



1.2.3 工作原理

步骤 1 和步骤 2 创建了一个 C# 的简单的命令行应用程序，使用了 .Net Framework 4.0 版本。步骤 3 引用了命名空间 System.Threading，该命名空间包含了该程序用到的所有类型。



正在执行中的程序实例可被称为一个进程。进程由一个或多个线程组成。这意味着当运行程序时，始终有一个执行程序代码的主线程。

步骤 4 中定义了方法 PrintNumbers，该方法会被主程序和新创建的线程使用。在步骤 5 中创建了一个线程来运行 PrintNumbers 方法。当我们构造一个线程时，ThreadStart 或 ParameterizedThreadStart 的实例委托会传给构造函数。我们只需指定在不同线程运行的方法名，而 C# 编译器则会在后台创建这些对象。然后我们在主线程中以通常的方式启动了一个线程来运行 PrintNumbers 方法。

结果两组范围为 1 到 10 的数字会随机交叉输出。这说明了 PrintNumbers 方法同时运行在主线程和另一个线程中。

1.3 暂停线程

本节将展示如何让一个线程等待一段时间而不用消耗操作系统资源。

1.3.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe2 目录中。

1.3.2 实现方式

请执行以下步骤来了解如何暂停线程而不消耗操作系统资源：

1. 启动 Visual Studio 2012。创建一个新的 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void PrintNumbers()
{
  Console.WriteLine("Starting...");
  for (int i = 1; i < 10; i++)
  {
    Console.WriteLine(i);
  }
}
static void PrintNumbersWithDelay()
{
  Console.WriteLine("Starting...");
  for (int i = 1; i < 10; i++)
  {
    Thread.Sleep(TimeSpan.FromSeconds(2));
    Console.WriteLine(i);
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
Thread t = new Thread(PrintNumbersWithDelay);
t.Start();
PrintNumbers();
```

5. 运行程序。

1.3.3 工作原理

当程序运行时，会创建一个线程，该线程会执行 PrintNumbersWithDelay 方法中的代码。然后会立即执行 PrintNumbers 方法。关键之处在于在 PrintNumbersWithDelay 方法中加入了 Thread.Sleep 方法调用。这将导致线程执行该代码时，在打印任何数字之前会等待指定的时

间（本例中是 2 秒钟）。当线程处于休眠状态时，它会占用尽可能少的 CPU 时间。结果我们会发现通常后运行的 PrintNumbers 方法中的代码会比独立线程中的 PrintNumbersWithDelay 方法中的代码先执行。

1.4 线程等待

本节将展示如何让程序等待另一个线程中的计算完成，然后在代码中使用该线程的计算结果。使用 Thread.Sleep 行不通，因为并不知道执行计算需要花费的具体时间。

1.4.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe3 目录中。

1.4.2 实现方式

请执行以下步骤来了解如何让一个程序等待另一个线程中的计算完成，并随后使用该线程的计算结果：

1. 启动 Visual Studio 2012。创建一个新的 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void PrintNumbersWithDelay()
{
  Console.WriteLine("Starting...");
  for (int i = 1; i < 10; i++)
  {
    Thread.Sleep(TimeSpan.FromSeconds(2));
    Console.WriteLine(i);
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
Console.WriteLine("Starting...");
Thread t = new Thread(PrintNumbersWithDelay);
t.Start();
t.Join();
Console.WriteLine("Thread completed");
```

5. 运行程序。

1.4.3 工作原理

当程序运行时，启动了一个耗时较长的线程来打印数字，打印每个数字前要等待两秒。但我们在主程序中调用了 t.Join 方法，该方法允许我们等待直到线程 t 完成。当线程 t 完成时，主程序会继续运行。借助该技术可以实现在两个线程间同步执行步骤。第一个线程会等待另一个线程完成后再继续执行。第一个线程等待时是处于阻塞状态（正如 1.3 节中调用 Thread.Sleep 方法一样）。

1.5 终止线程

本节将讲述如何终止线程的执行。

1.5.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe4 目录中。

1.5.2 实现方式

请执行以下步骤来了解如何终止线程的执行：

1. 启动 Visual Studio 2012。创建一个新的 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void PrintNumbersWithDelay()
{
  Console.WriteLine("Starting...");
  for (int i = 1; i < 10; i++)
  {
    Thread.Sleep(TimeSpan.FromSeconds(2));
    Console.WriteLine(i);
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
Console.WriteLine("Starting program...");
Thread t = new Thread(PrintNumbersWithDelay);
t.Start();
Thread.Sleep(TimeSpan.FromSeconds(6));
t.Abort();
Console.WriteLine("A thread has been aborted");
Thread t = new Thread(PrintNumbers);
t.Start();
PrintNumbers();
```

5. 运行程序。

1.5.3 工作原理

当主程序和单独的数字打印线程运行时，我们等待 6 秒后对线程调用了 t.Abort 方法。这给线程注入了 ThreadAbortException 方法，导致线程被终结。这非常危险，因为该异常可以在任何时刻发生并可能彻底摧毁应用程序。另外，使用该技术也不一定总能终止线程。目标线程可以通过处理该异常并调用 Thread.ResetAbort 方法来拒绝被终止。因此并不推荐使用 Abort 方法来关闭线程。可优先使用一些其他方法，比如提供一个 CancellationToken 方法来取消线程的执行。在第 3 章中我们会讨论该方法。

1.6 检测线程状态

本节将描述一个线程可能会有哪些状态。获取线程是否已经启动或是否处于阻塞状态等相应信息是非常有用的。请注意由于线程是独立运行的，所以其状态可以在任何时候被改变。

1.6.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe5 目录中。

1.6.2 实现方式

请执行以下步骤来了解如何确定线程状态及获取线程相关的信息。

1. 启动 Visual Studio 2012。创建一个新的 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void DoNothing()
{
  Thread.Sleep(TimeSpan.FromSeconds(2));
}

static void PrintNumbersWithStatus()
{
  Console.WriteLine("Starting...");
  Console.WriteLine(Thread.CurrentThread
  .ThreadState.ToString());
  for (int i = 1; i < 10; i++)
  {
    Thread.Sleep(TimeSpan.FromSeconds(2));
    Console.WriteLine(i);
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
Console.WriteLine("Starting program...");
Thread t = new Thread(PrintNumbersWithStatus);
Thread t2 = new Thread(DoNothing);
Console.WriteLine(t.ThreadState.ToString());
t2.Start();
t.Start();
for (int i = 1; i < 30; i++)
{
  Console.WriteLine(t.ThreadState.ToString());
}
Thread.Sleep(TimeSpan.FromSeconds(6));
t.Abort();
Console.WriteLine("A thread has been aborted");
Console.WriteLine(t.ThreadState.ToString());
Console.WriteLine(t2.ThreadState.ToString());
```

5. 运行程序。

1.6.3 工作原理

当主程序启动时定义了两个不同的线程。一个将被终止，另一个则会成功完成运行。线程状态位于 Thread 对象的 ThreadState 属性中。ThreadState 属性是一个 C# 枚举对象。刚开始线程状态为 ThreadState.Unstarted。然后我们启动线程，并估计在一个周期为 30 次迭代的区间中，线程状态会从 ThreadState.Running 变为 ThreadState.WaitSleepJoin。



请注意始终可以通过 Thread.CurrentThread 静态属性获得当前 Thread 对象。

如果实际情况与以上不符，请增加迭代次数。终止第一个线程后，会看到现在该线程状

态为 ThreadState.Aborted。程序也有可能会打印出 ThreadState.AbortRequested 状态。这充分说明了同步两个线程的复杂性。请记住不要在程序中使用线程终止。我在这里使用它只是为了展示相应的线程状态。

最后可以看到第二个线程 t2 成功完成并且状态为 ThreadState.Stopped。另外还有一些其他的线程状态，但是要么已经被弃用，要么没有我们实验过的几种状态有用。

1.7 线程优先级

本节将描述线程优先级的几种不同的可能选项。线程优先级决定了该线程可占用多少 CPU 时间。

1.7.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe6 目录中。

1.7.2 实现方式

请执行以下步骤来了解线程优先级的工作方式：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Diagnostics;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void RunThreads()
{
  var sample = new ThreadSample();

  var threadOne = new Thread(sample.CountNumbers);
  threadOne.Name = "ThreadOne";
  var threadTwo = new Thread(sample.CountNumbers);
  threadTwo.Name = "ThreadTwo";

  threadOne.Priority = ThreadPriority.Highest;
  threadTwo.Priority = ThreadPriority.Lowest;
  threadOne.Start();
  threadTwo.Start();

  Thread.Sleep(TimeSpan.FromSeconds(2));
```

```
  sample.Stop();
}
class ThreadSample
{
  private bool _isStopped = false;
  public void Stop()
  {
    _isStopped = true;
  }

  public void CountNumbers()
  {
    long counter = 0;

    while (!_isStopped)
    {
      counter++;
    }

    Console.WriteLine("{0} with {1,11} priority " +
      "has a count = {2,13}",
      Thread.CurrentThread.Name,
      Thread.CurrentThread.Priority,
      counter.ToString("N0"));
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
Console.WriteLine("Current thread priority: {0}",
  Thread.CurrentThread.Priority);
Console.WriteLine("Running on all cores available");
RunThreads();
Thread.Sleep(TimeSpan.FromSeconds(2));
Console.WriteLine("Running on a single core");
Process.GetCurrentProcess().ProcessorAffinity = new
  IntPtr(1);
RunThreads();
```

5. 运行程序。

1.7.3 工作原理

当主程序启动时定义了两个不同的线程。第一个线程优先级为 ThreadPriority.Highest，即具有最高优先级。第二个线程优先级为 ThreadPriority.Lowest，即具有最低优先级。我们先打印出主线程的优先级值，然后在所有可用的 CPU 核心上启动这两个线程。如果拥有一个以上的计算核心，将在两秒钟内得到初步结果。最高优先级的线程通常会计算更多的迭代，但是两个值应该很接近。然而，如果有其他程序占用了所有的 CPU 核心运行负载，结果则

会截然不同。

为了模拟该情形，我们设置了 ProcessorAffinity 选项，让操作系统将所有的线程运行在单个 CPU 核心（第一个核心）上。现在结果完全不同，并且计算耗时将超过 2 秒钟。这是因为 CPU 核心大部分时间在运行高优先级的线程，只留给剩下的线程很少的时间来运行。

请注意这是操作系统使用线程优先级的一个演示。通常你无需使用这种行为编写程序。

1.8 前台线程和后台线程

本节将描述前台线程和后台线程，及如何设置该选项来影响程序的行为。

1.8.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe7 目录中。

1.8.2 实现方式

请执行以下步骤来了解程序中前台线程和后台线程的效果：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
class ThreadSample
{
  private readonly int _iterations;

  public ThreadSample(int iterations)
  {
    _iterations = iterations;
  }
  public void CountNumbers()
  {
    for (int i = 0; i < _iterations; i++)
    {
      Thread.Sleep(TimeSpan.FromSeconds(0.5));
      Console.WriteLine("{0} prints {1}",
        Thread.CurrentThread.Name, i);
    }
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
var sampleForeground = new ThreadSample(10);
var sampleBackground = new ThreadSample(20);

var threadOne = new Thread(sampleForeground.CountNumbers);
threadOne.Name = "ForegroundThread";
var threadTwo = new Thread(sampleBackground.CountNumbers);
threadTwo.Name = "BackgroundThread";
threadTwo.IsBackground = true;

threadOne.Start();
threadTwo.Start();
```

5. 运行程序。

1.8.3 工作原理

当主程序启动时定义了两个不同的线程。默认情况下，显式创建的线程是前台线程。通过手动的设置 threadTwo 对象的 IsBackground 属性为 ture 来创建一个后台线程。通过配置来实现第一个线程会比第二个线程先完成。然后运行程序。

第一个线程完成后，程序结束并且后台线程被终结。这是前台线程与后台线程的主要区别：进程会等待所有的前台线程完成后再结束工作，但是如果只剩下后台线程，则会直接结束工作。

一个重要注意事项是如果程序定义了一个不会完成的前台线程，主程序并不会正常结束。

1.9 向线程传递参数

本节将描述如何提供一段代码来使用要求的数据运行另一个线程。我们将介绍不同的方式来满足此任务，并且回顾常见的错误。

1.9.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe8 目录中。

1.9.2 实现方式

请执行以下步骤来了解如何给一个线程传递参数：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void Count(object iterations)
{
  CountNumbers((int)iterations);
}

static void CountNumbers(int iterations)
{
  for (int i = 1; i <= iterations; i++)
  {
    Thread.Sleep(TimeSpan.FromSeconds(0.5));
    Console.WriteLine("{0} prints {1}",
      Thread.CurrentThread.Name, i);
  }
}
static void PrintNumber(int number)
{
  Console.WriteLine(number);
}

class ThreadSample
{
  private readonly int _iterations;

  public ThreadSample(int iterations)
  {
    _iterations = iterations;
  }
  public void CountNumbers()
  {
    for (int i = 1; i <= _iterations; i++)
    {
      Thread.Sleep(TimeSpan.FromSeconds(0.5));
      Console.WriteLine("{0} prints {1}",
        Thread.CurrentThread.Name, i);
    }
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
var sample = new ThreadSample(10);

var threadOne = new Thread(sample.CountNumbers);
threadOne.Name = "ThreadOne";
threadOne.Start();
threadOne.Join();
Console.WriteLine("--------------------------");
```

```
var threadTwo = new Thread(Count);
threadTwo.Name = "ThreadTwo";
threadTwo.Start(8);
threadTwo.Join();
Console.WriteLine("------------------------");

var threadThree = new Thread(() => CountNumbers(12));
threadThree.Name = "ThreadThree";
threadThree.Start();
threadThree.Join();
Console.WriteLine("------------------------");

int i = 10;
var threadFour = new Thread(() => PrintNumber(i));
i = 20;
var threadFive = new Thread(() => PrintNumber(i));
threadFour.Start();
threadFive.Start();
```

5. 运行程序。

1.9.3 工作原理

当主程序启动时，首先创建了 ThreadSample 类的一个对象，并提供了一个迭代次数。然后使用该对象的 CountNumbers 方法启动线程。该方法运行在另一个线程中，但是使用数字 10，该数字是通过 ThreadSample 对象的构造函数传入的。因此，我们只是使用相同的间接方式将该迭代次数传递给另一个线程。

1.9.4 更多信息

另一种传递数据的方式是使用 Thread.Start 方法。该方法会接收一个对象，并将该对象传递给线程。为了应用该方法，在线程中启动的方法必须接受 object 类型的单个参数。在创建 threadTwo 线程时演示了该方式。我们将 8 作为一个对象传递给了 Count 方法，然后 Count 方法被转换为整型。

接下来的方式是使用 lambda 表达式。lambda 表达式定义了一个不属于任何类的方法。我们创建了一个方法，该方法使用需要的参数调用了另一个方法，并在另一个线程中运行该方法。当启动 threadThree 线程时，打印出了 12 个数字，这正是我们通过 lambda 表达式传递的数字。

使用 lambda 表达式引用另一个 C# 对象的方式被称为闭包。当在 lambda 表达式中使用任何局部变量时，C# 会生成一个类，并将该变量作为该类的一个属性。所以实际上该方式与 threadOne 线程中使用的一样，但是我们无须定义该类，C# 编译器会自动帮我们实现。

这可能会导致几个问题。例如，如果在多个 lambda 表达式中使用相同的变量，它们会共享该变量值。在前一个例子中演示了这种情况。当启动 threadFour 和 threadFive 线程时，

它们都会打印 20，因为在这两个线程启动之前变量被修改为 20。

1.10 使用 C# 中的 lock 关键字

本节将描述如何确保当一个线程使用某些资源时，同时其他线程无法使用该资源。我们将了解该情况的必要性及整个线程安全概念都包含什么。

1.10.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe9 目录中。

1.10.2 实现方式

请执行以下步骤来了解如何使用 C# 中的 lock 关键字：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void TestCounter(CounterBase c)
{
  for (int i = 0; i < 100000; i++)
  {
    c.Increment();
    c.Decrement();
  }
}

class Counter : CounterBase
{
  public int Count { get; private set; }
  public override void Increment()
  {
    Count++;
  }

  public override void Decrement()
  {
    Count--;
  }
}

class CounterWithLock : CounterBase
{
```

```
  private readonly object _syncRoot = new Object();

  public int Count { get; private set; }

  public override void Increment()
  {
    lock (_syncRoot)
    {
      Count++;
    }
  }

  public override void Decrement()
  {
    lock (_syncRoot)
    {
      Count--;
    }
  }
}

abstract class CounterBase
{
  public abstract void Increment();
  public abstract void Decrement();
}
```

4. 在 Main 方法中加入以下代码片段：

```
Console.WriteLine("Incorrect counter");

var c = new Counter();

var t1 = new Thread(() => TestCounter(c));
var t2 = new Thread(() => TestCounter(c));
var t3 = new Thread(() => TestCounter(c));
t1.Start();
t2.Start();
t3.Start();
t1.Join();
t2.Join();
t3.Join();

Console.WriteLine("Total count: {0}",c.Count);
Console.WriteLine("--------------------------");
Console.WriteLine("Correct counter");

var c1 = new CounterWithLock();

t1 = new Thread(() => TestCounter(c1));
t2 = new Thread(() => TestCounter(c1));
t3 = new Thread(() => TestCounter(c1));
t1.Start();
t2.Start();
```

```
t3.Start();
t1.Join();
t2.Join();
t3.Join();
Console.WriteLine("Total count: {0}", c1.Count);
```

5. 运行程序。

1.10.3 工作原理

当主程序启动时，创建了一个 Counter 类的对象。该类定义了一个可以递增和递减的简单的计数器。然后我们启动了三个线程。这三个线程共享同一个 counter 实例，在一个周期中进行一次递增和一次递减。这将导致不确定的结果。如果运行程序多次，则会打印出多个不同的计数器值。结果可能是 0，但大多数情况下则不是 0。

这是因为 Counter 类并不是线程安全的。当多个线程同时访问 counter 对象时，第一个线程得到的 counter 值 10 并增加为 11。然后第二个线程得到的值是 11 并增加为 12。第一个线程得到 counter 值 12，但是递减操作发生前，第二个线程得到的 counter 值也是 12。然后第一个线程将 12 递减为 11 并保存回 counter 中，同时第二个线程进行了同样的操作。结果我们进行了两次递增操作但是只有一次递减操作，这显然不对。这种情形被称为竞争条件（race condition）。竞争条件是多线程环境中非常常见的导致错误的原因。

为了确保不会发生以上情形，必须保证当有线程操作 counter 对象时，所有其他线程必须等待直到当前线程完成操作。我们可以使用 lock 关键字来实现这种行为。如果锁定了一个对象，需要访问该对象的所有其他线程则会处于阻塞状态，并等待直到该对象解除锁定。这可能会导致严重的性能问题，在第 2 章中将会进一步学习该知识点。

1.11 使用 Monitor 类锁定资源

本节演示了另一个常见的多线程错误，被称为死锁（deadlock）。由于死锁将导致程序停止工作，以下例子中的第一部分使用了新的 Monitor 类来避免死锁。之前描述过的 lock 关键字则用于创建死锁。

1.11.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe10 目录中。

1.11.2 实现方式

请执行以下步骤来了解多线程中的死锁情况：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void LockTooMuch(object lock1, object lock2)
{
  lock (lock1)
  {
    Thread.Sleep(1000);
    lock (lock2);
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
object lock1 = new object();
object lock2 = new object();

new Thread(() => LockTooMuch(lock1, lock2)).Start();

lock (lock2)
{
  Thread.Sleep(1000);
  Console.WriteLine("Monitor.TryEnter allows not to get
    stuck, returning false after a specified timeout is
    elapsed");
  if (Monitor.TryEnter(lock1, TimeSpan.FromSeconds(5)))
  {
    Console.WriteLine("Acquired a protected resource
      succesfully");
  }
  else
  {
    Console.WriteLine("Timeout acquiring a resource!");
  }
}
new Thread(() => LockTooMuch(lock1, lock2)).Start();

Console.WriteLine("-----------------------------------");
lock (lock2)
{
  Console.WriteLine("This will be a deadlock!");
  Thread.Sleep(1000);
  lock (lock1)
  {
    Console.WriteLine("Acquired a protected resource
      succesfully");
  }
}
```

5. 运行程序。

1.11.3 工作原理

先看看 LockTooMuch 方法。在该方法中我们先锁定了第一个对象，等待一秒后锁定了第二个对象。然后在另一个线程中启动该方法。最后尝试在主线程中先后锁定第二个和第一个对象。

如果像该示例的第二部分一样使用 lock 关键字，将会造成死锁。第一个线程保持对 lock1 对象的锁定，等待直到 lock2 对象被释放。主线程保持对 lock2 对象的锁定并等待直到 lock1 对象被释放，但 lock1 对象永远不会被释放。

实际上 lock 关键字是 Monitor 类用例的一个语法糖。如果我们分解使用了 lock 关键字的代码，将会看到它如下面代码片段所示：

```
bool acquiredLock = false;
try
{
  Monitor.Enter(lockObject, ref acquiredLock);

// Code that accesses resources that are protected by the lock.

}
finally
{
  if (acquiredLock)
  {
    Monitor.Exit(lockObject);
  }
}
```

因此，我们可以直接使用 Monitor 类。其拥有 TryEnter 方法，该方法接受一个超时参数。如果在我们能够获取被 lock 保护的资源之前，超时参数过期，则该方法会返回 false。

1.12 处理异常

本节讲述了在线程中如何正确地处理异常。在线程中始终使用 try/catch 代码块是非常重要的，因为不可能在线程代码之外来捕获异常。

1.12.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码

放置在 BookSamples\Chapter1\Recipe11 目录中。

1.12.2 实现方式

请执行以下步骤来了解如何在线程中处理异常：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void BadFaultyThread()
{
  Console.WriteLine("Starting a faulty thread...");
  Thread.Sleep(TimeSpan.FromSeconds(2));
  throw new Exception("Boom!");
}

static void FaultyThread()
{
  try
  {
    Console.WriteLine("Starting a faulty thread...");
    Thread.Sleep(TimeSpan.FromSeconds(1));
    throw new Exception("Boom!");
  }
  catch (Exception ex)
  {
    Console.WriteLine("Exception handled: {0}",
      ex.Message);
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
var t = new Thread(FaultyThread);
t.Start();
t.Join();

try
{
  t = new Thread(BadFaultyThread);
  t.Start();
}
catch (Exception ex)
{
  Console.WriteLine("We won't get here!");
}
```

5. 运行程序。

1.12.3 工作原理

当主程序启动时，定义了两个将会抛出异常的线程。其中一个对异常进行了处理，另一个则没有。可以看到第二个异常没有被包裹启动线程的 try/catch 代码块捕获到。所以如果直接使用线程，一般来说不要在线程中抛出异常，而是在线程代码中使用 try/catch 代码块。

在较老版本的 .NET Framework 中（1.0 和 1.1），该行为是不一样的，未被捕获的异常不会强制应用程序关闭。可以通过添加一个包含以下代码片段的应用程序配置文件（比如 app.config）来使用该策略。

```
<configuration>
  <runtime>
    <legacyUnhandledExceptionPolicy enabled="1" />
  </runtime>
</configuration>
```



第 2 章 Chapter 2

线程同步

在本章中我们将讲述一些关于在多线程中使用共享资源的常用技术。你将学到以下内容：

- 执行基本的原子操作
- 使用 Mutex 类
- 使用 SemaphoreSlim 类
- 使用 AutoResetEvent 类
- 使用 ManualResetEventSlim 类
- 使用 CountDownEvent 类
- 使用 Barrier 类
- 使用 ReaderWriterLockSlim 类
- 使用 SpinWait 类

2.1 简介

正如第 1 章中所看到的一样，多个线程同时使用共享对象会造成很多问题。同步这些线程使得对共享对象的操作能够以正确的顺序执行是非常重要的。在 1.10 节，我们遇到了一个叫作竞争条件的问题。导致这问题的原因是多线程的执行并没有正确同步。当一个线程执行递增和递减操作时，其他线程需要依次等待。这种常见问题通常被称为**线程同步**。

有多种方式来实现线程同步。首先，如果无须共享对象，那么就无须进行线程同步。令人惊奇的是大多数时候可以通过重新设计程序来除移共享状态，从而去掉复杂的同步构造。请尽可能避免在多个线程间使用单一对象。

如果必须使用共享的状态，第二种方式是只使用**原子**操作。这意味着一个操作只占用一个量子的时间，一次就可以完成。所以只有当前操作完成后，其他线程才能执行其他操作。因此，你无须实现其他线程等待当前操作完成，这就避免了使用锁，也排除了死锁的情况。

如果上面的方式不可行，并且程序的逻辑更加复杂，那么我们不得不使用不同的方式来协调线程。方式之一是将等待的线程置于**阻塞**状态。当线程处于阻塞状态时，只会占用尽可能少的 CPU 时间。然而，这意味着将引入至少一次所谓的**上下文切换**（context switch）。上下文切换是指操作系统的线程调度器。该调度器会保存等待的线程的状态，并切换到另一个线程，依次恢复等待的线程的状态。这需要消耗相当多的资源。然而，如果线程要被挂起很长时间，那么这样做是值得的。这种方式又被称为**内核模式**（kernel-mode），因为只有操作系统的内核才能阻止线程使用 CPU 时间。

万一线程只需要等待一小段时间，最好只是简单的等待，而不用将线程切换到阻塞状态。虽然线程等待时会浪费 CPU 时间，但我们节省了上下文切换耗费的 CPU 时间。该方式又被称为**用户模式**（user-mode）。该方式非常轻量，速度很快，但如果线程需要等待较长时间则会浪费大量的 CPU 时间。

为了利用好这两种方式，可以使用**混合模式**（hybrid）。混合模式先尝试使用用户模式等待，如果线程等待了足够长的时间，则会切换到阻塞状态以节省 CPU 资源。

在本章中我们将介绍线程同步这一知识点。我们将讲解如何执行原子操作，以及如何使用 .NET framework 中现有的同步方式。

2.2 执行基本的原子操作

本节将展示如何对对象执行基本的原子操作，从而不用阻塞线程就可避免竞争条件。

2.2.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 7644_Code\Chapter2\Recipe1 目录中。

2.2.2 实现方式

请执行以下的步骤来了解基本的原子操作：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void TestCounter(CounterBase c)
{
  for (int i = 0; i < 100000; i++)
  {
    c.Increment();
    c.Decrement();
  }
}

class Counter : CounterBase
{
  private int _count;
  public int Count { get { return _count; } }

  public override void Increment()
  {
    _count++;
  }

  public override void Decrement()
  {
    _count--;
  }
}

class CounterNoLock : CounterBase
{
  private int _count;

  public int Count { get { return _count; } }

  public override void Increment()
  {
    Interlocked.Increment(ref _count);
  }

  public override void Decrement()
  {
    Interlocked.Decrement(ref _count);
  }
}

abstract class CounterBase
```

```
{
  public abstract void Increment();

  public abstract void Decrement();
}
```

4. 在 Main 方法中加入以下代码片段：

```
Console.WriteLine("Incorrect counter");

var c = new Counter();

var t1 = new Thread(() => TestCounter(c));
var t2 = new Thread(() => TestCounter(c));
var t3 = new Thread(() => TestCounter(c));
t1.Start();
t2.Start();
t3.Start();
t1.Join();
t2.Join();
t3.Join();
Console.WriteLine("Total count: {0}", c.Count);
Console.WriteLine("------------------------");
Console.WriteLine("Correct counter");

var c1 = new CounterNoLock();

t1 = new Thread(() => TestCounter(c1));
t2 = new Thread(() => TestCounter(c1));
t3 = new Thread(() => TestCounter(c1));
t1.Start();
t2.Start();
t3.Start();
t1.Join();
t2.Join();
t3.Join();
Console.WriteLine("Total count: {0}", c1.Count);
```

5. 运行程序。

2.2.3 工作原理

当程序运行时，会创建三个线程来运行 TestCounter 方法中的代码。该方法对一个对象按序执行了递增或递减操作。起初的 Counter 对象不是线程安全的，我们会遇到竞争条件。所以第一个例子中计数器的结果值是不确定的。我们可能会得到数字 0。然而如果运行程序多次，你将最终得到一些不正确的非零结果。

在第 1 章中，我们通过锁定对象解决了这个问题。在一个线程获取旧的计数器值并计算后赋予新的值之前，其他线程都被阻塞了。然而，如果我们采用上述方式执行该操作，

中途不能停止。而借助于Interlocked类，我们无需锁定任何对象即可获取到正确的结果。Interlocked提供了Increment、Decrement和Add等基本数学操作的原子方法，从而帮助我们在编写Counter类时无需使用锁。

2.3 使用Mutex类

本节将描述如何使用Mutex类来同步两个单独的程序。Mutex是一种原始的同步方式，其只对一个线程授予对共享资源的独占访问。

2.3.1 准备工作

为了学习本节，你需要安装Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在7644_Code\Chapter2\Recipe2目录中。

2.3.2 实现方式

请执行以下步骤来了解如何使用Mutex类实现两个单独的程序的同步：

1. 启动Visual Studio 2012。新建一个C#命令行应用程序项目。

2. 在Program.cs文件中加入以下using指令：

```
using System;
using System.Threading;
```

3. 在Main方法中加入以下代码片段：

```
const string MutexName = "CSharpThreadingCookbook";

using (var m = new Mutex(false, MutexName))
{
  if (!m.WaitOne(TimeSpan.FromSeconds(5), false))
  {
    Console.WriteLine("Second instance is running!");
  }
  else
  {
    Console.WriteLine("Running!");
    Console.ReadLine();
    m.ReleaseMutex();
  }
}
```

4. 运行程序。

2.3.3 工作原理

当主程序启动时，定义了一个指定名称的互斥量，设置initialOwner标志为false。这意

味着如果互斥量已经被创建，则允许程序获取该互斥量。如果没有获取到互斥量，程序则简单地显示 Running，等待直到按下了任何键，然后释放该互斥量并退出。

如果再运行同样一个程序，则会在 5 秒钟内尝试获取互斥量。如果此时在第一个程序中按下了任何键，第二个程序则会开始执行。然而，如果保持等待 5 秒钟，第二个程序将无法获取到该互斥量。



请注意具名的互斥量是全局的操作系统对象！请务必正确关闭互斥量。最好是使用 using 代码块来包裹互斥量对象。

该方式可用于在不同的程序中同步线程，可被推广到大量的使用场景中。

2.4 使用 SemaphoreSlim 类

本节将展示 SemaphoreSlim 类是如何作为 Semaphore 类的轻量级版本的。该类限制了同时访问同一个资源的线程数量。

2.4.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter2\Recipe3 目录中。

2.4.2 实现方式

请执行以下步骤来了解如何借助 SemaphoreSlim 类来限制访问同一个资源的线程数量：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。
2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static SemaphoreSlim _semaphore = new SemaphoreSlim(4);

static void AccessDatabase(string name, int seconds)
{
  Console.WriteLine("{0} waits to access a database", name);
  _semaphore.Wait();
  Console.WriteLine("{0} was granted an access to a database",
```

```
    name);
  Thread.Sleep(TimeSpan.FromSeconds(seconds));
  Console.WriteLine("{0} is completed", name);
  _semaphore.Release();

}
```

4. 在 Main 方法中加入以下代码片段：

```
for (int i = 1; i <= 6; i++)
{
  string threadName = "Thread " + i;
  int secondsToWait = 2 + 2*i;
  var t = new Thread(() => AccessDatabase(threadName,
    secondsToWait));
  t.Start();
}
```

5. 运行程序。

2.4.3 工作原理

当主程序启动时，创建了 SemaphoreSlim 的一个实例，并在其构造函数中指定允许的并发线程数量。然后启动了 6 个不同名称和不同初始运行时间的线程。

每个线程都尝试获取数据库的访问，但是我们借助于信号系统限制了访问数据库的并发数为 4 个线程。当有 4 个线程获取了数据库的访问后，其他两个线程需要等待，直到之前线程中的某一个完成工作并调用 _semaphore.Release 方法来发出信号。

2.4.4 更多信息

这里我们使用了混合模式，其允许我们在等待时间很短的情况下无需使用上下文切换。然而，有一个叫作 Semaphore 的 SemaphoreSlim 类的老版本。该版本使用纯粹的内核时间（kernel-time）方式。一般没必要使用它，除非是非常重要的场景。我们可以创建一个具名的 semaphore，就像一个具名的 mutex 一样，从而在不同的程序中同步线程。SemaphoreSlim 并不使用 Windows 内核信号量，而且也不支持进程间同步。所以在跨程序同步的场景下可以使用 Semaphore。

2.5 使用 AutoResetEvent 类

本节中的示例借助于 AutoResetEvent 类来从一个线程向另一个线程发送通知。AutoResetEvent 类可以通知等待的线程有某事件发生。

2.5.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 7644_Code\Chapter2\Recipe4 目录中。

2.5.2 实现方式

请执行以下步骤来了解如何借助 AutoResetEvent 类从一个线程中向另一个线程发送通知：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
private static AutoResetEvent _workerEvent = new
  AutoResetEvent(false);
private static AutoResetEvent _mainEvent = new
  AutoResetEvent(false);

static void Process(int seconds)
{
  Console.WriteLine("Starting a long running work...");
  Thread.Sleep(TimeSpan.FromSeconds(seconds));
  Console.WriteLine("Work is done!");
  _workerEvent.Set();
  Console.WriteLine("Waiting for a main thread to complete
    its work");
  _mainEvent.WaitOne();
  Console.WriteLine("Starting second operation...");
  Thread.Sleep(TimeSpan.FromSeconds(seconds));
  Console.WriteLine("Work is done!");
  _workerEvent.Set();
}
```

4. 在 Main 方法中加入以下代码片段：

```
var t = new Thread(() => Process(10));
t.Start();

Console.WriteLine("Waiting for another thread to complete
  work");
_workerEvent.WaitOne();
Console.WriteLine("First operation is completed!");
Console.WriteLine("Performing an operation on a main
  thread");
Thread.Sleep(TimeSpan.FromSeconds(5));
_mainEvent.Set();
Console.WriteLine("Now running the second operation on a
  second thread");
_workerEvent.WaitOne();
Console.WriteLine("Second operation is completed!");
```

5. 运行程序。

2.5.3 工作原理

当主程序启动时，定义了两个 AutoResetEvent 实例。其中一个是从子线程向主线程发信号，另一个实例是从主线程向子线程发信号。我们向 AutoResetEvent 构造方法传入 false，定义了这两个实例的初始状态为 unsignaled。这意味着任何线程调用这两个对象中的任何一个的 WaitOne 方法将会被阻塞，直到我们调用了 Set 方法。如果初始事件状态为 true，那么 AutoResetEvent 实例的状态为 signaled，如果线程调用 WaitOne 方法则会被立即处理。然后事件状态自动变为 unsignaled，所以需要再对该实例调用一次 Set 方法，以便让其他的线程对该实例调用 WaitOne 方法从而继续执行。

然后我们创建了第二个线程，其会执行第一个操作 10 秒钟，然后等待从第二个线程发出的信号。该信号意味着第一个操作已经完成。现在第二个线程在等待主线程的信号。我们对主线程做了一些附加工作，并通过调用 _mainEvent.Set 方法发送了一个信号。然后等待从第二个线程发出的另一个信号。

AutoResetEvent 类采用的是内核时间模式，所以等待时间不能太长。使用 2.6 节中的 ManualResetEventslim 类更好，因为它使用的是混合模式。

2.6 使用 ManualResetEventSlim 类

本节将描述如何使用 ManualResetEventSlim 类来在线程间以更灵活的方式传递信号。

2.6.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter2\Recipe5 目录中。

2.6.2 实现方式

请执行以下步骤来了解 ManualResetEventSlim 类的使用方法：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static ManualResetEventSlim _mainEvent = new
ManualResetEventSlim(false);

static void TravelThroughGates(string threadName,
  int seconds)
{
  Console.WriteLine("{0} falls to sleep", threadName);
  Thread.Sleep(TimeSpan.FromSeconds(seconds));
  Console.WriteLine("{0} waits for the gates to open!",
    threadName);
  _mainEvent.Wait();
  Console.WriteLine("{0} enters the gates!", threadName);
}
```

4. 在 Main 方法中加入以下代码片段：

```
var t1 = new Thread(() => TravelThroughGates("Thread 1",
  5));
var t2 = new Thread(() => TravelThroughGates("Thread 2",
  6));
var t3 = new Thread(() => TravelThroughGates("Thread 3",
  12));
t1.Start();
t2.Start();
t3.Start();
Thread.Sleep(TimeSpan.FromSeconds(6));
Console.WriteLine("The gates are now open!");
_mainEvent.Set();
Thread.Sleep(TimeSpan.FromSeconds(2));
_mainEvent.Reset();
Console.WriteLine("The gates have been closed!");
Thread.Sleep(TimeSpan.FromSeconds(10));
Console.WriteLine("The gates are now open for the second
  time!");
_mainEvent.Set();
Thread.Sleep(TimeSpan.FromSeconds(2));
Console.WriteLine("The gates have been closed!");
_mainEvent.Reset();
```

5. 运行程序。

2.6.3 工作原理

当主程序启动时，首先创建了 ManualResetEventSlim 类的一个实例。然后启动了三个线程，等待事件信号通知它们继续执行。

ManualResetEvnetSlim 的整个工作方式有点像人群通过大门。2.5 节中讨论过的 AutoResetEvent 事件像一个旋转门，一次只允许一人通过。ManualResetEventSlim 是 ManualResetEvent 的混合版本，一直保持大门敞开直到手动调用 Reset 方法。当调用 _mainEvent.Set

时，相当于打开了大门从而允许准备好的线程接收信号并继续工作。然而线程 3 还处于睡眠状态，没有赶上时间。当调用 _mainEvent.Reset 相当于关闭了大门。最后一个线程已经准备好执行，但是不得不等待下一个信号，即要等待好几秒钟。

2.6.4 更多信息

在前面某小节中，我们使用了一种无法在操作系统层面工作的混合模式。如果我们需要全局事件，则可以使用 EventWaitHandle 类，其是 AutoResetEvent 和 ManualResetEvent 类的基类。

2.7 使用 CountDownEvent 类

本节将描述如何使用 CountdownEvent 信号类来等待直到一定数量的操作完成。

2.7.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter2\Recipe6 目录中。

2.7.2 实现方式

请执行以下步骤来了解 CountDownEvent 类的使用：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static CountdownEvent _countdown = new CountdownEvent(2);

static void PerformOperation(string message, int seconds)
{
  Thread.Sleep(TimeSpan.FromSeconds(seconds));
  Console.WriteLine(message);
  _countdown.Signal();
}
```

4. 在 Main 方法中加入以下代码片段：

```
Console.WriteLine("Starting two operations");
var t1 = new Thread(() => PerformOperation("Operation 1 is
```

```
  completed", 4));
var t2 = new Thread(() => PerformOperation("Operation 2 is
  completed", 8));
t1.Start();
t2.Start();
_countdown.Wait();
Console.WriteLine("Both operations have been completed.");
_countdown.Dispose();
```

5. 运行程序。

2.7.3 工作原理

当主程序启动时，创建了一个 CountdownEvent 实例，在其构造函数中指定了当两个操作完成时会发出信号。然后我们启动了两个线程，当它们执行完成后会发出信号。一旦第二个线程完成，主线程会从等待 CountdownEvent 的状态中返回并继续执行。针对需要等待多个异步操作完成的情形，使用该方式是非常便利的。

然而这有一个重大的缺点。如果调用 _countdown.Signal() 没达到指定的次数，那么 _countdown.Wait() 将一直等待。请确保使用 CountdownEvent 时，所有线程完成后都要调用 Signal 方法。

2.8 使用 Barrier 类

本节将展示另一种有意思的同步方式，被称为 Barrier。Barrier 类用于组织多个线程及时在某个时刻碰面。其提供了一个回调函数，每次线程调用了 SignalAndWait 方法后该回调函数会被执行。

2.8.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter2\Recipe7 目录中。

2.8.2 实现方式

请执行以下步骤来了解 Barrier 类的使用：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static Barrier _barrier = new Barrier(2,
  b => Console.WriteLine("End of phase {0}",
    b.CurrentPhaseNumber + 1));

static void PlayMusic(string name, string message,
  int seconds)
{
  for (int i = 1; i < 3; i++)
  {
    Console.WriteLine("-----------------------------------
      ----------");
    Thread.Sleep(TimeSpan.FromSeconds(seconds));
    Console.WriteLine("{0} starts to {1}", name, message);
    Thread.Sleep(TimeSpan.FromSeconds(seconds));
    Console.WriteLine("{0} finishes to {1}", name,
      message);
    _barrier.SignalAndWait();
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
var t1 = new Thread(() => PlayMusic("the guitarist",
  "play an amazing solo", 5));
var t2 = new Thread(() => PlayMusic("the singer",
  "sing his song", 2));

t1.Start();
t2.Start();
```

5. 运行程序。

2.8.3 工作原理

我们创建了 Barrier 类，指定了我们想要同步两个线程。在两个线程中的任何一个调用了 _barrier.SignalAndWait 方法后，会执行一个回调函数来打印出阶段。

每个线程将向 Barrier 发送两次信号，所以会有两个阶段。每次这两个线程调用 SignalAndWait 方法时，Barrier 将执行回调函数。这在多线程迭代运算中非常有用，可以在每个迭代结束前执行一些计算。当最后一个线程调用 SignalAndWait 方法时可以在迭代结束时进行交互。

2.9 使用 ReaderWriterLockSlim 类

本节将描述如何使用 ReaderWriterLockSlim 来创建一个线程安全的机制，在多线程中对一个集合进行读写操作。ReaderWriterLockSlim 代表了一个管理资源访问的锁，允许多个线程同时读取，以及独占写。

2.9.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter2\Recipe8 目录中。

2.9.2 实现方式

请执行以下步骤来了解如何使用 ReaderWriterLockSlim 类来创建一个线程安全的机制，在多线程中对集合进行读写操作：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Collections.Generic;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static ReaderWriterLockSlim _rw = new
  ReaderWriterLockSlim();
static Dictionary<int, int> _items =
  new Dictionary<int, int>();

static void Read()
{
  Console.WriteLine("Reading contents of a dictionary");
  while (true)
  {
    try
    {
      _rw.EnterReadLock();
      foreach (var key in _items.Keys)
      {
        Thread.Sleep(TimeSpan.FromSeconds(0.1));
      }
    }
    finally
    {
      _rw.ExitReadLock();
    }
  }
}

static void Write(string threadName)
{
  while (true)
  {
    try
    {
      int newKey = new Random().Next(250);
      _rw.EnterUpgradeableReadLock();
      if (!_items.ContainsKey(newKey))
```

```
        {
          try
          {
            _rw.EnterWriteLock();
            _items[newKey] = 1;
            Console.WriteLine("New key {0} is added to a
              dictionary by a {1}", newKey, threadName);
          }
          finally
          {
            _rw.ExitWriteLock();
          }
        }
        Thread.Sleep(TimeSpan.FromSeconds(0.1));
      }
      finally
      {
        _rw.ExitUpgradeableReadLock();
      }
    }
  }
```

4. 在 Main 方法中加入以下代码片段：

```
new Thread(Read){ IsBackground = true }.Start();
new Thread(Read){ IsBackground = true }.Start();
new Thread(Read){ IsBackground = true }.Start();

new Thread(() => Write("Thread 1")){ IsBackground =
  true }.Start();
new Thread(() => Write("Thread 2")){ IsBackground =
  true }.Start();

Thread.Sleep(TimeSpan.FromSeconds(30));
```

5. 运行程序。

2.9.3 工作原理

当主程序启动时，同时运行了三个线程来从字典中读取数据，还有另外两个线程向该字典中写入数据。我们使用 ReaderWriterLockSlim 类来实现线程安全，该类专为这样的场景而设计。

这里使用两种锁：读锁允许多线程读取数据，写锁在被释放前会阻塞了其他线程的所有操作。获取读锁时还有一个有意思的场景，即从集合中读取数据时，根据当前数据而决定是否获取一个写锁并修改该集合。一旦得到写锁，会阻止阅读者读取数据，从而浪费大量的时间，因此获取写锁后集合会处于阻塞状态。为了最小化阻塞浪费的时间，可以使用 EnterUpgradeableReadLock 和 ExitUpgradeableReadLock 方法。先获取读锁后读取数据。如果发现必须修改底层集合，只需使用 EnterWriteLock 方法升级锁，然后快速执行一次写操作，

最后使用 ExitWriteLock 释放写锁。

在本例中，我们先生成一个随机数。然后获取读锁并检查该数是否存在于字典的键集合中。如果不存在，将读锁更新为写锁然后将该新键加入到字典中。始终使用 tyr/finaly 代码块来确保在捕获锁后一定会释放锁，这是一项好的实践。

所有的线程都被创建为后台线程。主线程在所有后台线程完成后会等待 30 秒。

2.10 使用 SpinWait 类

本节将描述如何不使用内核模型的方式来使线程等待。另外，我们介绍了 SpinWait，它是一个混合同步构造，被设计为使用用户模式等待一段时间，然后切换到内核模式以节省 CPU 时间。

2.10.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter2\Recipe9 目录中。

2.10.2 实现方式

请执行以下步骤来了解如何不借助于内核模式方法来实现线程等待：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static volatile bool _isCompleted = false;

static void UserModeWait()
{
  while (!_isCompleted)
  {
    Console.Write(".");
  }
  Console.WriteLine();
  Console.WriteLine("Waiting is complete");
}

static void HybridSpinWait()
{
  var w = new SpinWait();
  while (!_isCompleted)
```

```
  {
    w.SpinOnce();
    Console.WriteLine(w.NextSpinWillYield);
  }
  Console.WriteLine("Waiting is complete");
}
```

4. 在 Main 方法中加入以下代码片段：

```
var t1 = new Thread(UserModeWait);
var t2 = new Thread(HybridSpinWait);

Console.WriteLine("Running user mode waiting");
t1.Start();
Thread.Sleep(20);
_isCompleted = true;
Thread.Sleep(TimeSpan.FromSeconds(1));
_isCompleted = false;
Console.WriteLine("Running hybrid SpinWait construct
  waiting");
t2.Start();
Thread.Sleep(5);
_isCompleted = true;
```

5. 运行程序。

2.10.3 工作原理

当主程序启动时，定义了一个线程，将执行一个无止境的循环，直到 20 毫秒后主线程设置 _isCompleted 变量为 true。我们可以试验运行该周期为 20 ~ 30 秒，通过 Windows 任务管理器测量 CPU 的负载情况。取决于 CPU 内核数量，任务管理器将显示一个显著的处理时间。

我们使用 volatile 关键字来声明 _isCompleted 静态字段。Volatile 关键字指出一个字段可能会被同时执行的多个线程修改。声明为 volatile 的字段不会被编译器和处理器优化为只能被单个线程访问。这确保了该字段总是最新的值。

然后我们使用了 SpinWait 版本，用于在每个迭代打印一个特殊标志位来显示线程是否切换为阻塞状态。运行该线程 5 毫秒来查看结果。刚开始，SpinWait 尝试使用用户模式，在 9 个迭代后，开始切换线程为阻塞状态。如果尝试测量该版本的 CPU 负载，在 Windows 任务管理器将不会看到任何 CPU 的使用。



Chapter 3 第 3 章

使用线程池

在本章中，我们将描述多线程中使用共享资源的常用技术。你将学到以下内容：

- 在线程池中调用委托
- 向线程池中放入异步操作
- 线程池与并行度
- 实现一个取消选项
- 在线程池中使用等待事件处理器及超时
- 使用计时器
- 使用 BackgroundWorker 组件

3.1 简介

在之前的章节中我们讨论了创建线程和线程协作的几种方式。现在考虑另一种情况，即只花费极少的时间来完成创建很多异步操作。正如在第 1 章的**简介**小节中讨论过的一样，创建线程是昂贵的操作，所以为每个短暂的异步操作创建线程会产生显著的开销。

为了解决该问题，有一个常用的方式叫做**池**（pooling）。线程池可以成功地适应于任何需要大量短暂的开销大的资源的情形。我们事先分配一定的资源，将这些资源放入到资源池。每次需要新的资源，只需从池中获取一个，而不用创建一个新的。当该资源不再被使用时，就将其返回到池中。

.NET 线程池是该概念的一种实现。通过 System.Threading.ThreadPool 类型可以使用线

程池。线程池是受 .NET **通用语言运行时**（Common Language Runtime，简称 CLR）管理的。这意味着每个 CLR 都有一个线程池实例。ThreadPool 类型拥有一个 QueueUserWorkItem 静态方法。该静态方法接受一个**委托**，代表用户自定义的一个异步操作。在该方法被调用后，委托会进入到内部队列中。如果池中没有任何线程，将创建一个新的**工作线程**（worker thread）并将队列中第一个委托放入到该工作线程中。

如果想线程池中放入新的操作，当之前的所有操作完成后，很可能只需重用一个线程来执行这些新的操作。然而，如果放置新的操作过快，线程池将创建更多的线程来执行这些操作。创建太多的线程是有限制的，在这种情况下新的操作将在队列中等待直到线程池中的工作线程有能力来执行它们。



保持线程中的操作都是短暂的是非常重要的。不要在线程池中放入长时间运行的操作，或者阻塞工作线程。这将导致所有工作线程变得繁忙，从而无法服务用户操作。这会导致性能问题和非常难以调试的错误。

当停止向线程池中放置新操作时，线程池最终会删除一定时间后过期的不再使用的线程。这将释放所有那些不再需要的系统资源。

我想再次强调线程池的用途是执行运行时间短的操作。使用线程池可以减少并行度耗费及节省操作系统资源。我们只使用较少的线程，但是以比平常更慢的速度来执行异步操作，使用一定数量的可用的工作线程批量处理这些操作。如果操作能快速地完成则比较适用线程池，但是执行长时间运行的计算密集型操作则会降低性能。

另一个重要事情是在 ASP.NET 应用程序中使用线程池时要相当小心。ASP.NET 基础设施使用自己的线程池，如果在线程池中浪费所有的工作线程，Web 服务器将不能够服务新的请求。在 ASP.NET 中只推荐使用输入 / 输出密集型的异步操作，因为其使用了一个不同的方式，叫做 **I/O 线程**。我们将在第 9 章中讨论 I/O 线程。



请注意线程池中的工作线程都是后台线程。这意味着当所有的前台线程（包括主程序线程）完成后，所有的后台线程将停止工作。

在本章中，我们将学习使用线程池来执行异步操作。本章将覆盖将操作放入线程池的不同方式，以及如何取消一个操作，并防止其长时间运行。

3.2　在线程池中调用委托

本节将展示在线程池中如何异步的执行委托。另外，我们将讨论一个叫做**异步编程模型**（Asynchronous Programming Model，简称 APM）的方式，这是 .NET 历史中第一个异步编程模式。

3.2.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter3\Recipe1 目录中。

3.2.2 实现方式

请执行以下步骤来了解如何在线程池中调用委托：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
private delegate string RunOnThreadPool(out int threadId);

private static void Callback(IAsyncResultar)
{
  Console.WriteLine("Starting a callback...");
  Console.WriteLine("State passed to a callback: {0}",
    ar.AsyncState);
  Console.WriteLine("Is thread pool thread: {0}",
    Thread.CurrentThread.IsThreadPoolThread);
  Console.WriteLine("Thread pool worker thread id: {0}",
    Thread.CurrentThread.ManagedThreadId);
}

private static string Test(out intthreadId)
{
  Console.WriteLine("Starting...");
  Console.WriteLine("Is thread pool thread: {0}",
    Thread.CurrentThread.IsThreadPoolThread);
  Thread.Sleep(TimeSpan.FromSeconds(2));
  threadId = Thread.CurrentThread.ManagedThreadId;
  return string.Format("Thread pool worker thread id was:
    {0}", threadId);
}
```

4. 在 Main 方法中加入以下代码片段：

```
int threadId = 0;

RunOnThreadPool poolDelegate = Test;

var t = new Thread(() => Test(out threadId));
t.Start();
t.Join();
```

```
Console.WriteLine("Thread id: {0}", threadId);

IAsyncResult r = poolDelegate.BeginInvoke(out threadId,
  Callback, "a delegate asynchronous call");
r.AsyncWaitHandle.WaitOne();

string result = poolDelegate.EndInvoke(out threadId, r);

Console.WriteLine("Thread pool worker thread id: {0}",
  threadId);
Console.WriteLine(result);

Thread.Sleep(TimeSpan.FromSeconds(2));
```

5. 运行程序。

3.2.3 工作原理

当程序运行时，使用旧的方式创建了一个线程，然后启动它并等待完成。由于线程的构造函数只接受一个无任何返回结果的方法，我们使用了 **lambda 表达式**来将对 Test 方法的调用包起来。我们通过打印出 Thread. CurrentThread.IsThreadPoolThread 属性值来确保该线程不是来自线程池。我们也打印出了受管理的线程 ID 来识别代码是被哪个线程执行的。

然后定义了一个委托并调用 BeginInvoke 方法来运行该委托。BeginInvoke 方法接受一个回调函数。该回调函数会在异步操作完成后会被调用，并且一个用户自定义的状态会传给该回调函数。该状态通常用于区分异步调用。结果，我们得到了一个实现了 IAsyncResult 接口的 result 对象。BeginInvoke 立即返回了结果，当线程池中的工作线程在执行异步操作时，仍允许我们继续其他工作。当需要异步操作的结果时，可以使用 BeginInvoke 方法调用返回的 result 对象。我们可以使用 result 对象的 IsCompleted 属性轮询结果。但是在本例子中，使用的是 AsyncWaitHandle 属性来等待直到操作完成。当操作完成后，会得到一个结果，可以通过委托调用 EndInvoke 方法，将 IAsyncResult 对象传递给委托参数。



事实上使用 AsyncWaitHandle 并不是必要的。如果注释掉 r.AsyncWaitHandle.WaitOne，代码照样可以成功运行，因为 EndInvoke 方法事实上会等待异步操作完成。调用 EndInvoke 方法（或者针对其他异步 API 的 EndOperationName 方法）是非常重要的，因为该方法会将任何未处理的异常抛回到调用线程中。当使用这种异步 API 时，请确保始终调用了 Begin 和 End 方法。

当操作完成后，传递给 BeginInvoke 方法的回调函数将被放置到线程池中，确切地说是一个工作线程中。如果在 Main 方法定义的结尾注释掉 Thread.Sleep 方法调用，回调函数将不会被执行。这是因为当主线程完成后，所有的后台线程会被停止，包括该回调函数。对委托

和回调函数的异步调用很可能会被同一个工作线程执行。通过工作线程 ID 可以容易地看出。

使用 BeginOperationName/EndOperationName 方法和 .NET 中的 IAsyncResult 对象等方式被称为异步编程模型（或 APM 模式），这样的方法对被称为异步方法。该模式也被应用于多个 .NET 类库的 API 中，但在现代编程中，更推荐使用**任务并行库**（Task Parallel Library，简称 TPL）来组织异步 API。第 4 章将会讨论该主题。

3.3 向线程池中放入异步操作

本节将描述如何向线程池中放入异步操作。

3.3.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter3\Recipe2 目录中。

3.3.2 实现方式

请执行以下步骤来了解如何向线程池中放入异步操作：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
usingSystem.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
private static void AsyncOperation(object state)
{
  Console.WriteLine("Operation state: {0}",
    state ?? "(null)");
  Console.WriteLine("Worker thread id: {0}",
    Thread.CurrentThread.ManagedThreadId);
  Thread.Sleep(TimeSpan.FromSeconds(2));
}
```

4. 在 Main 方法中加入以下代码片段：

```
const int x = 1;
const int y = 2;
const string lambdaState = "lambda state 2";

ThreadPool.QueueUserWorkItem(AsyncOperation);
Thread.Sleep(TimeSpan.FromSeconds(1));
```

```
ThreadPool.QueueUserWorkItem(AsyncOperation,
  "async state");
Thread.Sleep(TimeSpan.FromSeconds(1));

ThreadPool.QueueUserWorkItem( state => {
  Console.WriteLine("Operation state: {0}", state);
  Console.WriteLine("Worker thread id: {0}",
    Thread.CurrentThread.ManagedThreadId);
  Thread.Sleep(TimeSpan.FromSeconds(2));
}, "lambda state");

ThreadPool.QueueUserWorkItem( _ => {
  Console.WriteLine("Operation state: {0}, {1}", x+y,
    lambdaState);
  Console.WriteLine("Worker thread id: {0}",
    Thread.CurrentThread.ManagedThreadId);
  Thread.Sleep(TimeSpan.FromSeconds(2));
}, "lambda state");

Thread.Sleep(TimeSpan.FromSeconds(2));
```

5. 运行程序。

3.3.3　工作原理

首先定义了 AsyncOperation 方法，其接受单个 object 类型的参数。然后使用 QueueUserWorkItem 方法将该方法放到线程池中。接着再次放入该方法，但是这次给方法调用传入了一个状态对象。该对象将作为状态参数传递给 AsynchronousOperation 方法。

在操作完成后让线程睡眠一秒钟，从而让线程池拥有为新操作重用线程的可能性。如果注释掉所有的 Thread.Sleep 调用，那么所有打印出的线程 ID 多半是不一样的。如果 ID 是一样的，那很可能是前两个线程被重用来运行接下来的两个操作。

首先将一个 lambda 表达式放置到线程池中。这里没什么特别的。我们使用了 labmbda 表达式语法，从而无须定义一个单独的方法。

然后，我们使用**闭包**机制，从而无须传递 lambda 表达式的状态。闭包更灵活，允许我们向异步操作传递一个以上的对象而且这些对象具有静态类型。所以之前介绍的传递对象给方法回调的机制既冗余又过时。在 C# 中有了闭包后就不再需要使用它了。

3.4　线程池与并行度

本节将展示线程池如何工作于大量的异步操作，以及它与创建大量单独的线程的方式有何不同。

3.4.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter3\Recipe3 目录中。

3.4.2 实现方式

请执行以下步骤来了解线程池如何工作于大量的异步操作，以及它与创建大量单独的线程的方式的不同之处：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Diagnostics;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void UseThreads(int numberOfOperations)
{
  using (var countdown = new CountdownEvent(
    numberOfOperations)) {

    Console.WriteLine("Scheduling work by creating
      threads");
    for (int i=0; i<numberOfOperations; i++) {
      var thread = new Thread(() => {
        Console.Write("{0},", Thread.CurrentThread.
          ManagedThreadId);
        Thread.Sleep(TimeSpan.FromSeconds(0.1));
        countdown.Signal();
      });
      thread.Start();
    }
    countdown.Wait();
    Console.WriteLine();
  }
}
static void UseThreadPool(int numberOfOperations)
{
  using (var countdown = new CountdownEvent(
    numberOfOperations)) {

    Console.WriteLine("Starting work on a threadpool");
    for (int i=0; i<numberOfOperations; i++) {
      ThreadPool.QueueUserWorkItem( _ => {
        Console.Write("{0},", Thread.CurrentThread.
          ManagedThreadId);
        Thread.Sleep(TimeSpan.FromSeconds(0.1));
        countdown.Signal();
      });
```

```
    }
    countdown.Wait();
    Console.WriteLine();
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
const int numberOfOperations = 500;
var sw = new Stopwatch();
sw.Start();
UseThreads(numberOfOperations);
sw.Stop();
Console.WriteLine("Execution time using threads: {0}",
  sw.ElapsedMilliseconds);

sw.Reset();
sw.Start();
UseThreadPool(numberOfOperations);
sw.Stop();
Console.WriteLine("Execution time using threads: {0}",
  sw.ElapsedMilliseconds);
```

5. 运行程序。

3.4.3　工作原理

当主程序启动时，创建了很多不同的线程，每个线程都运行一个操作。该操作打印出线程 ID 并阻塞线程 100 毫秒。结果我们创建了 500 个线程，全部并行运行这些操作。虽然在我的机器上的总耗时是 300 毫秒，但是所有线程消耗了大量的操作系统资源。

然后我们使用了执行同样的任务，只不过不为每个操作创建一个线程，而将它们放入到线程池中。然后线程池开始执行这些操作。线程池在快结束时创建更多的线程，但是仍然花费了更多的时间，在我机器上是 12 秒。我们为操作系统节省了内存和线程数，但是为此付出了更长的执行时间。

3.5　实现一个取消选项

本节将通过一个示例来展示如何在线程池中取消异步操作。

3.5.1　准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter3\Recipe4 目录中。

3.5.2 实现方式

请执行以下步骤来了解如何在线程中实现一个取消选项：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void AsyncOperation1(CancellationToken token) {
  Console.WriteLine("Starting the first task");
  for (int i=0; i<5; i++) {
    if (token.IsCancellationRequested) {
      Console.WriteLine("The first task has been
        canceled.");
      return;
    }
    Thread.Sleep(TimeSpan.FromSeconds(1));
  }
  Console.WriteLine("The first task has completed
    succesfully");
}

static void AsyncOperation2(CancellationToken token) {
  try {
    Console.WriteLine("Starting the second task");

    for (int i=0; i<5; i++) {
      token.ThrowIfCancellationRequested();
      Thread.Sleep(TimeSpan.FromSeconds(1));
    }
    Console.WriteLine("The second task has completed
      successfully");
  }
  catch (OperationCanceledException) {
    Console.WriteLine("The second task has been
      canceled.");
  }
}

private static void AsyncOperation3(
  CancellationToken token) {

  boolcancellationFlag = false;
  token.Register(()=>cancellationFlag=true);
  Console.WriteLine("Starting the third task");
  for (int i=0; i<5; i++) {
    if (cancellationFlag) {
      Console.WriteLine("The third task has been
        canceled.");
      return;
    }
```

```
    Thread.Sleep(TimeSpan.FromSeconds(1));
  }
  Console.WriteLine("The third task has completed
    succesfully");
}
```

4. 在 Main 方法中加入以下代码片段：

```
using (var cts = new CancellationTokenSource()) {
  CancellationToken token = cts.Token;
  ThreadPool.QueueUserWorkItem(_ => AsyncOperation1(
    token));
  Thread.Sleep(TimeSpan.FromSeconds(2));
  cts.Cancel();
}

using (var cts = new CancellationTokenSource()) {
  CancellationToken token = cts.Token;
  ThreadPool.QueueUserWorkItem(_ => AsyncOperation2(
   token));
  Thread.Sleep(TimeSpan.FromSeconds(2));
  cts.Cancel();
}

using (var cts = new CancellationTokenSource()) {
  CancellationToken token = cts.Token;
  ThreadPool.QueueUserWorkItem(_ => AsyncOperation3(
    token));
  Thread.Sleep(TimeSpan.FromSeconds(2));
  cts.Cancel();
}

Thread.Sleep(TimeSpan.FromSeconds(2));
```

5. 运行程序。

3.5.3　工作原理

本节中介绍了 CancellationTokenSource 和 CancellationToken 两个新类。它们在 .NET4.0 被引入，目前是实现异步操作的取消操作的事实标准。由于线程池已经存在了很长时间，并没有特殊的 API 来实现取消标记功能，但是仍然可以对线程池使用上述 API。

在本程序中使用了三种方式来实现取消过程。第一个是轮询来检查 CancellationToken.IsCancellationRequested 属性。如果该属性为 true，则说明操作需要被取消，我们必须放弃该操作。

第二种方式是抛出一个 OperationCancelledException 异常。这允许在操作之外控制取消过程，即需要取消操作时，通过操作之外的代码来处理。

最后一种方式是注册一个回调函数。当操作被取消时，在线程池将调用该回调函数。这允许链式传递一个取消逻辑到另一个异步操作中。

3.6 在线程池中使用等待事件处理器及超时

本节将描述如何在线程池中对操作实现超时，以及如何在线程池中正确地等待。

3.6.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter3\Recipe5 目录中。

3.6.2 实现方式

请执行以下步骤来了解在线程池中实现超时及正确的等待：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void RunOperations(TimeSpanworkerOperationTimeout) {
  using (var evt = new ManualResetEvent(false))
  using (var cts = new CancellationTokenSource()) {
    Console.WriteLine("Registering timeout operations...");
    var worker = ThreadPool.RegisterWaitForSingleObject(
      evt, (state, isTimedOut) => WorkerOperationWait(cts,
        isTimedOut), null, workerOperationTimeout, true);

    Console.WriteLine("Starting long running
      operation...");

    ThreadPool.QueueUserWorkItem(_ => WorkerOperation(cts.Token,
evt));

    Thread.Sleep(workerOperationTimeout.Add(TimeSpan.
      FromSeconds(2)));
    worker.Unregister(evt);
  }
}

static void WorkerOperation(CancellationToken token,
  ManualResetEventevt) {

  for(int i=0; i<6; i++) {
    if (token.IsCancellationRequested) {
      return;
    }
    Thread.Sleep(TimeSpan.FromSeconds(1));
  }
  evt.Set();
```

```
}

static void WorkerOperationWait(CancellationTokenSource cts
  bool isTimedOut) {

  if (isTimedOut) {
    cts.Cancel();
    Console.WriteLine("Worker operation timed out and was
      canceled.");
  }
  else {
    Console.WriteLine("Worker operation succeded.");
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
RunOperations(TimeSpan.FromSeconds(5));
RunOperations(TimeSpan.FromSeconds(7));
```

5. 运行程序。

3.6.3　工作原理

线程池还有一个有用的方法：ThreadPool.RegisterWaitForSingleObject。该方法允许我们将回调函数放入线程池中的队列中。当提供的等待事件处理器收到信号或发生超时时，该回调函数将被调用。这允许我们为线程池中的操作实现超时功能。

首先按顺序向线程池中放入一个耗时长的操作。它运行 6 秒钟然后一旦成功完成，会设置一个 ManualResetEvent 信号类。其他的情况下，比如需要取消操作，则该操作会被丢弃。

然后我们注册了第二个异步操作。当从 ManualResetEvent 对象接受到一个信号后，该异步操作会被调用。如果第一个操作顺利完成，会设置该信号量。另一种情况是第一个操作还未完成就已经超时。如果发生了该情况，我们会使用 CancellationToken 来取消第一个操作。

最后，为操作提供 5 秒的超时时间是不够的。这是因为操作会花费 6 秒来完成，只能取消该操作。所以如果提供 7 秒的超时时间是可行的，该操作会顺利完成。

3.6.4　更多信息

当有大量的线程必须处于阻塞状态中等待一些多线程事件发信号时，以上方式非常有用。借助于线程池的基础设施，我们无需阻塞所有这样的线程。可以释放这些线程直到信号事件被设置。在服务器端应用程序中这是个非常重要的应用场景，因为服务器端应用程序要求高伸缩性及高性能。

3.7 使用计时器

本节将描述如何使用 System.Threading.Timer 对象来在线程池中创建周期性调用的异步操作。

3.7.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter3\Recipe6 目录中。

3.7.2 实现方式

请执行以下步骤来了解如何在线程池中创建周期性调用的异步操作：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static Timer _timer;

static void TimerOperation(DateTime start) {
  TimeSpan elapsed = DateTime.Now - start;
  Console.WriteLine("{0} seconds from {1}. Timer thread
    pool thread id: {2}", elapsed.Seconds, start,
    Thread.CurrentThread.ManagedThreadId);
}
```

4. 在 Main 方法中加入以下代码片段：

```
Console.WriteLine("Press 'Enter' to stop the timer...");
DateTime start = DateTime.Now;
_timer = new Timer(_ => TimerOperation(start), null,
  TimeSpan.FromSeconds(1), TimeSpan.FromSeconds(2));

Thread.Sleep(TimeSpan.FromSeconds(6));

_timer.Change(TimeSpan.FromSeconds(1),
  TimeSpan.FromSeconds(4));

Console.ReadLine();

_timer.Dispose();
```

5. 运行程序。

3.7.3　工作原理

我们首先创建了一个 Timer 实例。第一个参数是一个 lambda 表达式，将会在线程池中被执行。我们调用 TimerOperation 方法并给其提供一个起始时间。由于无须使用用户状态对象，所以第二个参数为 null。然后指定了什么时候会第一次运行 TimerOperation，以及之后再次调用的间隔时间。所以第一个值实际上说明一秒后会启动第一次操作，然后每隔两秒再次运行。

之后等待 6 秒后修改计时器。在调用 _timer.Change 方法一秒后启动 TimerOperation，然后每隔 4 秒再次运行。



计时器还可以更复杂！

可以以更复杂的方式使用计时器。比如，可以通过 Timeout.Infinet 值提供给计时器一个间隔参数来只允许计时器操作一次。然后在计时器异步操作内，能够设置下一次计时器操作将被执行的时间。具体时间取决于自定义业务逻辑。

最后等待回车键被按下以结束整个应用程序。在程序运行过程中可以看到自从程序启动后经历的时间。

3.8　使用 BackgroundWorker 组件

本节实例演示了另一种异步编程的方式，即使用 BackgroundWorker 组件。

借助于该对象，可以将异步代码组织为一系列事件及事件处理器。你将学会如何使用该组件进行异步编程。

3.8.1　准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter3\Recipe7 目录中。

3.8.2　实现方式

请执行以下步骤来学习如何使用 BackgroundWorker 组件：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.ComponentModel;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void Worker_DoWork(object sender, DoWorkEventArgs e)
{
  Console.WriteLine("DoWork thread pool thread id: {0}",
    Thread.CurrentThread.ManagedThreadId);
  var bw = (BackgroundWorker) sender;
  for (int i=1; i<=100; i++) {

    if (bw.CancellationPending) {
      e.Cancel = true;
      return;
    }

    if (i%10 == 0) {
      bw.ReportProgress(i);
    }

    Thread.Sleep(TimeSpan.FromSeconds(0.1));
  }
  e.Result = 42;
}

static void Worker_ProgressChanged(object sender,
  ProgressChangedEventArgs e){

  Console.WriteLine("{0}% completed. Progress thread pool
    thread id: {1}", e.ProgressPercentage, Thread.
      CurrentThread.ManagedThreadId);
}

static void Worker_Completed(object sender,
  RunWorkerCompletedEventArgs e) {

  Console.WriteLine("Completed thread pool thread id: {0}",
    Thread.CurrentThread.ManagedThreadId);
  if (e.Error != null) {
    Console.WriteLine("Exception {0} has occured.",
      e.Error.Message);
  }
  else if (e.Cancelled) {
    Console.WriteLine("Operation has been canceled.");
  }
  else {
    Console.WriteLine("The answer is: {0}", e.Result);
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
var bw = new BackgroundWorker();
bw.WorkerReportsProgress = true;
bw.WorkerSupportsCancellation = true;

bw.DoWork += Worker_DoWork;
bw.ProgressChanged += Worker_ProgressChanged;
```

```
bw.RunWorkerCompleted += Worker_Completed;

bw.RunWorkerAsync();

Console.WriteLine("Press C to cancel work");
do {
  if (Console.ReadKey(true).KeyChar == 'C') {
    bw.CancelAsync();
  }

}
while(bw.IsBusy);
```

5. 运行程序。

3.8.3　工作原理

当程序启动时，创建了一个 BackgroundWorker 组件的实例。显式地指出该后台工作线程支持取消操作及该操作进度的通知。

接下来是最有意思的部分。我们没有使用线程池和委托，而是使用了另一个 C# 语法，称为**事件**。事件表示了一些通知的**源**或当通知到达时会有所响应的一系列**订阅者**。在本例中，我们将订阅三个事件，当这些事件发生时，将调用相应的**事件处理器**。当事件通知其订阅者时，具有特殊的定义签名的方法将被调用。

因此，除了将异步 API 组织为 Begin/End 方法对，还可以只启动一个异步操作然后订阅给不同的事件。这些事件在该操作执行时会被触发。这种方式被称为**基于事件的异步模式**（Event-based Asynchronous Pattern，简称 EAP）。这是历史上第二种用来构造异步程序的方式，现在更推荐使用 TPL，第 4 章中将会描述该方式。

我们共定义了三个事件。第一个是 oWork 事件。当一个后台工作对象通过 RunWorkerAsync 方法启动一个异步操作时，该事件处理器将被调用。该事件处理器将会运行在线程池中。如果需要取消操作，则这里是主要的操作点来取消执行。同时也可以提供该操作的运行进程信息。最后，得到结果后，将结果设置给事件参数，然后 RunWorkerCompleted 事件处理器将被调用。在该方法中，可以知道操作是成功完成，还是发生错误，抑或被取消。

基于此，BackgroundWorker 组件实际上被使用于 **Windows 窗体应用程序**（Windows Forms Applications，简称 WPF）中。该实现通过后台工作事件处理器的代码可以直接与 UI 控制器交互。与线程池中的线程与 UI 控制器交互的方式相比较，使用 BackgroundWorker 组件的方式更加自然和好用。



Chapter 4 第 4 章

使用任务并行库

在本章中，我们将学习一个新的异步编程范式，即任务并行库。你将学到以下知识：

- 创建任务
- 使用任务执行基本的操作
- 组合任务
- 将 APM 模式转换为任务
- 将 EAP 模式转换为任务
- 实现取消选项
- 处理任务中的异常
- 并行运行任务
- 使用 TaskScheduler 配置任务的执行

4.1 简介

我们在之前的章节中学习了什么是线程，如何使用线程，以及为什么需要线程池。使用线程池可以使我们在减少并行度花销时节省操作系统资源。我们可以认为线程池是一个**抽象层**，其向程序员隐藏了使用线程的细节，使我们专心处理程序逻辑，而不是各种线程问题。

然而使用线程池也相当复杂。从线程池的工作线程中获取结果并不容易。我们需要实现自定义方式来获取结果，而且万一有异常发生，还需将异常正确地传播到初始线程中。除此

以外，创建一组相关的异步操作，以及实现当前操作执行完成后下一操作才会执行的逻辑也不容易。

在尝试解决这些问题的过程中，创建了异步编程模型及基于事件的异步模式。在**第 3 章**中提到过基于事件的异步模式。这些模式使得获取结果更容易，传播异常也更轻松，但是组合多个异步操作仍需大量工作，需要编写大量的代码。

为了解决所有的问题，.Net Framework4.0 引入了一个新的关于异步操作的 API。它叫做**任务并行库**（Task Parallel Library，简称 TPL）。.Net Framework 4.5 版对该 API 进行了轻微的改进，使用更简单。在本书的项目中将使用最新版的 TPL，即 .Net Framework 4.5 版中的 API。TPL 可被认为是线程池之上的又一个抽象层，其对程序员隐藏了与线程池交互的底层代码，并提供了更方便的细粒度的 API。

TPL 的核心概念是任务。一个任务代表了一个异步操作，该操作可以通过多种方式运行，可以使用或不使用独立线程运行。在本章中将探究任务的所有使用细节。



默认情况下，程序员无须知道任务实际上是如何执行的。TPL 通过向用户隐藏任务的实现细节从而创建一个抽象层。遗憾的是，有些情况下这会导致诡秘的错误，比如试图获取任务的结果时程序被挂起。本章有助于理解 TPL 底层的原理，以及如何避免不恰当的使用方式。

一个任务可以通过多种方式和其他任务组合起来。例如，可以同时启动多个任务，等待所有任务完成，然后运行一个任务对之前所有任务的结果进行一些计算。TPL 与之前的模式相比，其中一个关键优势是其具有用于组合任务的便利的 API。

处理任务中的异常结果有多种方式。由于一个任务可能会由多个其他任务组成，这些任务也可能依次拥有各自的子任务，所以有一个 AggregateException 的概念。这种异常可以捕获底层任务内部的所有异常，并允许单独处理这些异常。

而且，最后但并不是最不重要的，C# 5.0 已经内置了对 TPL 的支持，允许我们使用新的 await 和 async 关键字以平滑的、舒服的方式操作任务。在第 5 章中将讨论该主题。

在本章中我们将学习使用 TPL 来执行异步操作。我们将学习什么是任务，如何用不同的方式创建任务，以及如何将任务组合在一起。我们会讨论如何将遗留的 APM 和 EAP 模式转换为使用任务，还有如何正确地处理异常，如何取消任务，以及如何使多个任务同时执行。另外，还将讲述如何在 Windows GUI 应用程序中正确地使用任务。

4.2　创建任务

本节展示了任务的基本概念。你将学习如何创建和执行任务。

4.2.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter4\Recipe1 目录中。

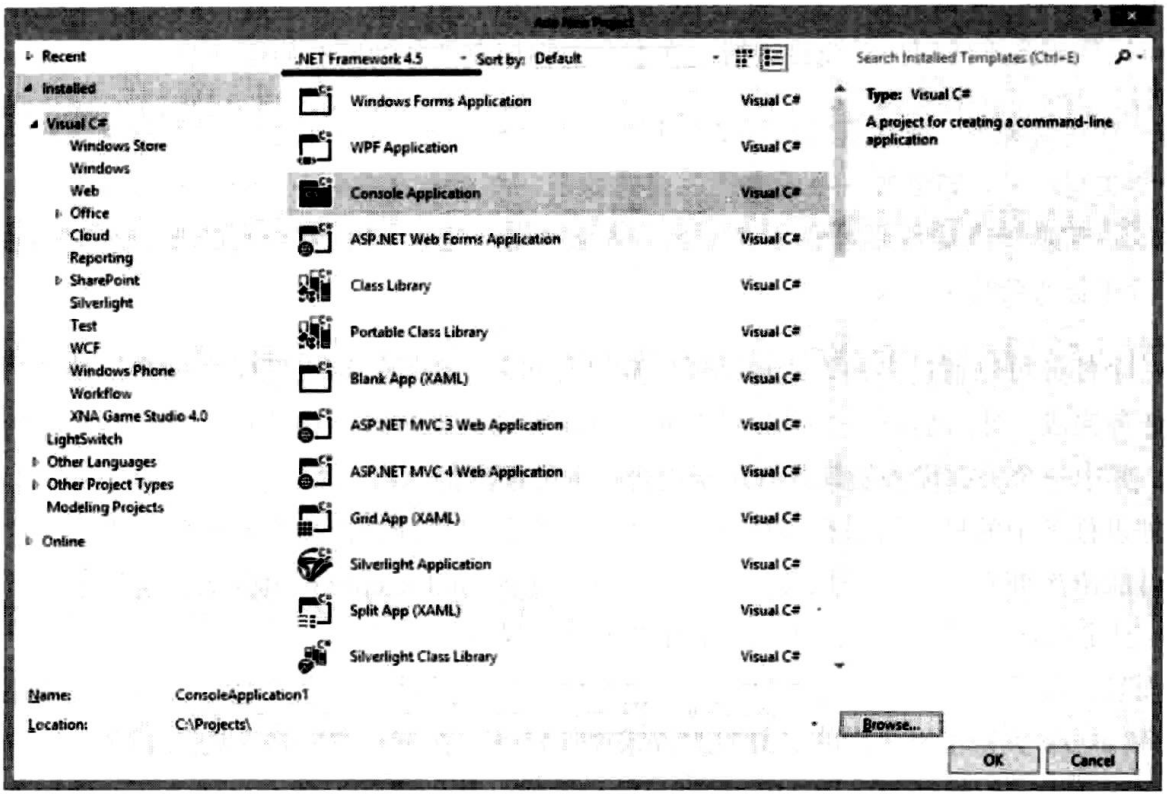
4.2.2 实现方式

请执行以下步骤来创建和执行任务：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。



这里请确定使用的版本为 .Net Framework 4.5。从现在开始我们将在所有项目中使用该版本。



2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void TaskMethod(string name){
  Console.WriteLine("Task {0} is running on a thread id
    {1}. Is thread pool thread: {2}", name,
```

```
        Thread.CurrentThread.ManagedThreadId,
          Thread.CurrentThread.IsThreadPoolThread);
}
```

4. 在 Main 方法中加入以下代码片段：

```
var t1 = new Task(() =>TaskMethod("Task 1"));
var t2 = new Task(() =>TaskMethod("Task 2"));
t2.Start();
t1.Start();
Task.Run(() =>TaskMethod("Task 3"));
Task.Factory.StartNew(() => TaskMethod("Task 4"));
Task.Factory.StartNew(() => TaskMethod("Task 5"),
  TaskCreationOptions.LongRunning);
Thread.Sleep(TimeSpan.FromSeconds(1));
```

5. 运行程序。

4.2.3　工作原理

当程序运行时，我们使用 Task 的构造函数创建了两个任务。我们传入一个 lambda 表达式作为 Action 委托。这可以使我们给 TaskMethod 提供一个 string 参数。然后使用 Start 方法运行这些任务。



请注意只有调用了这些任务的 Start 方法，才会执行任务。很容易忘记真正启动任务。

然后使用 Task.Run 和 Task.Factory.StartNew 方法来运行了另外两个任务。与使用 Task 构造函数的不同之处在于这两个被创建的任务会立即开始工作，所以无需显式地调用这些任务的 Start 方法。从 Task 1 到 Task 4 的所有任务都被放置在线程池的工作线程中并以未指定的顺序运行。如果多次运行该程序，就会发现任务的执行顺序是不确定的。

Task.Run 方法只是 Task.Factory.StartNew 的一个快捷方式，但是后者有附加的选项。通常如果无特殊需求，则可使用前一个方法，如 Task 5 所示。我们标记该任务为长时间运行，结果该任务将不会使用线程池，而在单独的线程中运行。然而，根据运行该任务的当前的**任务调度程序**（task scheduler），运行方式有可能不同。你将在本章的 4.10 节了解什么是任务调度程序。

4.3　使用任务执行基本的操作

本节将描述如何从任务中获取结果值。我们将通过几个场景来了解在线程池中和主线程中运行任务的不同之处。

4.3.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter4\Recipe2 目录中。

4.3.2 实现方式

请执行以下步骤来了解任务的基本的操作：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
static Task<int>CreateTask(string name){
  return new Task<int>(() =>TaskMethod(name));
}

static int TaskMethod(string name){
  Console.WriteLine("Task {0} is running on a thread id
    {1}. Is thread pool thread: {2}",name,
      Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread);
  Thread.Sleep(TimeSpan.FromSeconds(2));
  return 42;
}
```

4. 在 Main 方法中加入以下代码片段：

```
TaskMethod("Main Thread Task");
Task<int> task = CreateTask("Task 1");
task.Start();
int result = task.Result;
Console.WriteLine("Result is: {0}", result);

task = CreateTask("Task 2");
task.RunSynchronously();
result = task.Result;
Console.WriteLine("Result is: {0}", result);

task = CreateTask("Task 3");
task.Start();

while (!task.IsCompleted){
  Console.WriteLine(task.Status);
  Thread.Sleep(TimeSpan.FromSeconds(0.5));
```

```
}

Console.WriteLine(task.Status);
result = task.Result;
Console.WriteLine("Result is: {0}", result);
```

5. 运行程序。

4.3.3　工作原理

首先直接运行 TaskMethod 方法，这里并没有把它封装到一个任务中。结果根据它提供给我们的主线程的信息可以得知该方法是被同步执行的。很显然它不是线程池中的线程。

然后我们运行了 Task 1，使用 Start 方法启动该任务并等待结果。该任务会被放置在线程池中，并且主线程会等待，直到任务返回前一直处于阻塞状态。

Task 2 和 Task 1 类似，除了 Task 2 是通过 RunSynchronously() 方法运行的。该任务会运行在主线程中，该任务的输出与第一个例子中直接同步调用 TaskMethod 的输出完全一样。这是个非常好的优化，可以避免使用线程池来执行非常短暂的操作。

我们用以运行 Task 1 相同的方式来运行 Task 3。但这次没有阻塞主线程，只是在该任务完成前循环打印出任务状态。结果展示了多种任务状态，分别是 Creatd、Running 和 RanToCompletion。

4.4　组合任务

本节将展示如何设置相互依赖的任务。我们将学习如何创建一个任务，使其在父任务完成后才会被运行。另外，将探寻为非常短暂的任务节省线程开销的可能性。

4.4.1　准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter4\Recipe3 目录中。

4.4.2　实现方式

请执行以下步骤来组合任务：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
static int TaskMethod(string name, int seconds){
  Console.WriteLine("Task {0} is running on a thread id
    {1}. Is thread pool thread: {2}", name,
      Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread);
  Thread.Sleep(TimeSpan.FromSeconds(seconds));
  return 42 * seconds;
}
```

4. 在 Main 方法中加入以下代码片段：

```
var firstTask = new Task<int>(() =>TaskMethod("First Task",
  3));
var secondTask = new Task<int>(() =>TaskMethod("Second
  Task", 2));

firstTask.ContinueWith(
  t =>Console.WriteLine("The first answer is {0}. Thread id
    {1}, is thread pool thread: {2}", t.Result,
      Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread),
          TaskContinuationOptions.OnlyOnRanToCompletion);

firstTask.Start();
secondTask.Start();

Thread.Sleep(TimeSpan.FromSeconds(4));

Task continuation = secondTask.ContinueWith(
  t =>Console.WriteLine("The second answer is {0}. Thread
    id {1}, is thread pool thread: {2}", t.Result,
      Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread),
          TaskContinuationOptions.OnlyOnRanToCompletion |
            TaskContinuationOptions.ExecuteSynchronously);

continuation.GetAwaiter().OnCompleted(
  () =>Console.WriteLine("Continuation Task Completed!
    Thread id {0}, is thread pool thread: {1}",
      Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread));

Thread.Sleep(TimeSpan.FromSeconds(2));
Console.WriteLine();

firstTask = new Task<int>(() => {
  varinnerTask = Task.Factory.StartNew(() =>TaskMethod(
    "Second Task", 5), TaskCreationOptions.
      AttachedToParent);
  innerTask.ContinueWith(t =>TaskMethod("Third Task", 2),
    TaskContinuationOptions.AttachedToParent);
```

```
  return TaskMethod("First Task", 2);
});

firstTask.Start();

while (!firstTask.IsCompleted){
  Console.WriteLine(firstTask.Status);
  Thread.Sleep(TimeSpan.FromSeconds(0.5));
}
Console.WriteLine(firstTask.Status);

Thread.Sleep(TimeSpan.FromSeconds(10));
```

5. 运行程序。

4.4.3　工作原理

当主程序启动时，我们创建了两个任务，并为第一个任务设置了一个**后续操作**（continuation，一个代码块，会在当前任务完成后运行）。然后启动这两个任务并等待 4 秒，这个时间足够两个任务完成。然后给第二个任务运行另一个后续操作，并通过指定 TaskContinuationOptions.ExecuteSynchronously 选项来尝试同步执行该后续操作。如果后续操作耗时非常短暂，使用以上方式是非常有用的，因为放置在主线程中运行比放置在线程池中运行要快。可以实现这一点是因为第二个任务恰好在那刻完成。如果注释掉 4 秒的 Thread.Sleep 方法，将会看到该代码被放置到线程池中，这是因为还未从之前的任务中得到结果。

最后我们为之前的后续操作也定义了一个后续操作，但这里使用了一个稍微不同的方式，即使用了新的 GetAwaiter 和 OnCompleted 方法。这些方法是 C# 5.0 语言中异步机制中的方法。第 5 章中将会覆盖该主题。

本节示例的最后部分与父子线程有关。我们创建了一个新任务，当运行该任务时，通过提供一个 TaskCreationOptions.AttachedToParent 选项来运行一个所谓的子任务。



子任务必须在父任务运行时创建，并正确的附加给父任务！

这意味着只有所有子任务结束工作，父任务才会完成。通过提供一个 TaskContinuationOptions 选项也可以给在子任务上运行后续操作。该后续操作也会影响父任务，并且直到最后一个子任务结束它才会运行完成。

4.5　将 APM 模式转换为任务

本节将说明如何将过时的 APM API 转换为任务。多个示例覆盖了转换过程中可能发生

的不同情况。

4.5.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter4\Recipe4 目录中。

4.5.2 实现方式

请执行以下步骤将 APM 模式转换为任务：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
private delegate string AsynchronousTask(string
  threadName);
private delegate string IncompatibleAsynchronousTask(out
  int threadId);

private static void Callback(IAsyncResultar){
  Console.WriteLine("Starting a callback...");
  Console.WriteLine("State passed to a callback: {0}",
    ar.AsyncState);
  Console.WriteLine("Is thread pool thread: {0}",
    Thread.CurrentThread.IsThreadPoolThread);
  Console.WriteLine("Thread pool worker thread id: {0}",
    Thread.CurrentThread.ManagedThreadId);
}

private static string Test(string threadName){
  Console.WriteLine("Starting...");
  Console.WriteLine("Is thread pool thread: {0}",
    Thread.CurrentThread.IsThreadPoolThread);
  Thread.Sleep(TimeSpan.FromSeconds(2));
  Thread.CurrentThread.Name = threadName;
  return string.Format("Thread name: {0}",
    Thread.CurrentThread.Name);
}

private static string Test(out int threadId){
  Console.WriteLine("Starting...");
  Console.WriteLine("Is thread pool thread: {0}",
    Thread.CurrentThread.IsThreadPoolThread);
  Thread.Sleep(TimeSpan.FromSeconds(2));
```

```
  threadId = Thread.CurrentThread.ManagedThreadId;
  return string.Format("Thread pool worker thread id was:
    {0}", threadId);
}
```

4. 在 Main 方法中加入以下代码片段：

```
int threadId;
AsynchronousTask d = Test;
IncompatibleAsynchronousTask e = Test;

Console.WriteLine("Option 1");
Task<string> task = Task<string>.Factory.FromAsync(
  d.BeginInvoke("AsyncTaskThread", Callback, "a delegate
    asynchronous call"), d.EndInvoke);

task.ContinueWith(t =>Console.WriteLine("Callback is
  finished, now running a continuation! Result: {0}",
    t.Result));

while (!task.IsCompleted){
  Console.WriteLine(task.Status);
  Thread.Sleep(TimeSpan.FromSeconds(0.5));
}
Console.WriteLine(task.Status);
Thread.Sleep(TimeSpan.FromSeconds(1));

Console.WriteLine("----------------------------------------");
Console.WriteLine();
Console.WriteLine("Option 2");

task = Task<string>.Factory.FromAsync(
  d.BeginInvoke, d.EndInvoke, "AsyncTaskThread", "a
    delegate asynchronous call");
task.ContinueWith(t =>Console.WriteLine("Task is completed,
  now running a continuation! Result: {0}",
    t.Result));
while (!task.IsCompleted){
  Console.WriteLine(task.Status);
  Thread.Sleep(TimeSpan.FromSeconds(0.5));
}
Console.WriteLine(task.Status);
Thread.Sleep(TimeSpan.FromSeconds(1));

Console.WriteLine("----------------------------------------");
Console.WriteLine();
Console.WriteLine("Option 3");

IAsyncResult ar = e.BeginInvoke(out threadId, Callback, "a
  delegate asynchronous call");
ar = e.BeginInvoke(out threadId, Callback, "a delegate
  asynchronous call");
task = Task<string>.Factory.FromAsync(ar, _ =>e.EndInvoke(
  out threadId, ar));
task.ContinueWith(t =>
  Console.WriteLine("Task is completed, now running a
```

```
        continuation! Result: {0}, ThreadId: {1}",
          t.Result, threadId));

while (!task.IsCompleted){
  Console.WriteLine(task.Status);
  Thread.Sleep(TimeSpan.FromSeconds(0.5));
}
Console.WriteLine(task.Status);

Thread.Sleep(TimeSpan.FromSeconds(1));
```

5. 运行程序。

4.5.3 工作原理

这里我们定义了两种委托。其中一个使用了 out 参数，因此在将 APM 模式转换为任务时，与标准的 TPL API 是不兼容的。这样的转换有三个示例。

将 APM 转换为 TPL 的关键点是 Task<T>.Factory.FromAsync 方法，T 是异步操作结果的类型。该方法有数个重载。在第一个例子中传入了 IAsyncResult 和 Func<IAsyncResult，string>，这是一个将 IAsyncResult 的实现作为参数并返回一个字符串的方法。由于第一个委托类型提供的 EndMethod 与该签名是兼容的，所以将该委托的异步调用转换为任务没有任何问题。

第二个例子做的事与第一个非常相似，但是使用了不同的 FromAsync 方法重载，该重载并不允许指定一个将会在异步委托调用完成后被调用的回调函数。但我们可以使用后续操作替代它。但如果回调函数很重要，可以使用第一个例子所示的方法。

最后一个例子展示了一个小技巧。这次 IncompatibleAsynchronousTask 委托的 EndMethod 使用了 out 参数，与 FromAsync 方法重载并不兼容。然而，可以很容易地将 EndMethod 调用封装到一个 lambda 表达式中，从而适合任务工厂方法。

可以在等待异步操作结果过程中打印出任务状态，从而了解底层任务的运行情况。可以看到第一个任务的状态为 WaitingForActivation，这意味着 TPL 基础设施实际上还未启动该任务。

4.6 将 EAP 模式转换为任务

本节将描述如何将基于事件的异步操作转换为任务。在本节中，你将发现有一个可靠的模式可适用于 .Net Framework 类库中的所有基于事件的异步 API。

4.6.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter4\Recipe5 目录中。

4.6.2　实现方式

请执行以下步骤来将 EAP 模式转换为任务：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.ComponentModel;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
static int TaskMethod(string name, int seconds){
  Console.WriteLine("Task {0} is running on a thread id
    {1}. Is thread pool thread: {2}", name,
      Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread);
  Thread.Sleep(TimeSpan.FromSeconds(seconds));
  return 42 * seconds;
}
```

4. 在 Main 方法中加入以下代码片段：

```
var tcs = new TaskCompletionSource<int>();

var worker = new BackgroundWorker();
worker.DoWork += (sender, eventArgs) =>
{
  eventArgs.Result = TaskMethod("Background worker", 5);
};
worker.RunWorkerCompleted += (sender, eventArgs) =>{
  if (eventArgs.Error != null) {
    tcs.SetException(eventArgs.Error);
  }
  else if (eventArgs.Cancelled) {
    tcs.SetCanceled();
  }
    else {
      tcs.SetResult((int)eventArgs.Result);
    }
};

worker.RunWorkerAsync();

int result = tcs.Task.Result;

Console.WriteLine("Result is: {0}", result);
```

5. 运行程序。

4.6.3 工作原理

这是一个将 EAP 模式转换为任务的既简单又优美的示例。关键点在于使用 TaskCompletionSource<T> 类型，T 是异步操作结果类型。

不要忘记将 tcs.SetResult 调用封装在 try-catch 代码块中，从而保证错误信息始终会设置给任务完成源对象。也可以使用 TrySetResult 方法来替代 SetResult 方法，以保证结果能被成功设置。

4.7 实现取消选项

本节是关于如何给基于任务的异步操作实现取消流程。我们将学习如何正确的使用取消标志，以及在任务真正运行前如何得知其是否被取消。

4.7.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter4\Recipe6 目录中。

4.7.2 实现方式

请执行以下步骤来为基于任务的异步操作实现取消选项：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
private static int TaskMethod(string name, int seconds,
  CancellationToken token){

  Console.WriteLine("Task {0} is running on a thread id
    {1}. Is thread pool thread: {2}", name,
      Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread);
  for (int i = 0; i< seconds; i ++) {
    Thread.Sleep(TimeSpan.FromSeconds(1));
    if (token.IsCancellationRequested)
      return -1;
  }
  return 42*seconds;
}
```

4. 在 Main 方法中加入以下代码片段：

```
var cts = new CancellationTokenSource();
var longTask = new Task<int>(() =>TaskMethod("Task 1", 10,
  cts.Token), cts.Token);
Console.WriteLine(longTask.Status);
cts.Cancel();
Console.WriteLine(longTask.Status);
Console.WriteLine("First task has been cancelled before
  execution");
cts = new CancellationTokenSource();
longTask = new Task<int>(() =>TaskMethod("Task 2", 10,
  cts.Token), cts.Token);
longTask.Start();
for (int i = 0; i< 5; i++ ){
  Thread.Sleep(TimeSpan.FromSeconds(0.5));
  Console.WriteLine(longTask.Status);
}
cts.Cancel();
for (int i = 0; i< 5; i++){
  Thread.Sleep(TimeSpan.FromSeconds(0.5));
  Console.WriteLine(longTask.Status);
}

Console.WriteLine("A task has been completed with result
  {0}.", longTask.Result);
```

5. 运行程序。

4.7.3　工作原理

第 3 章中我们已经讨论了取消标志概念，你已经相当熟悉了。而本节又是一个关于为 TPL 任务实现取消选项的简单例子。

首先仔细看看 longTask 的创建代码。我们将给底层任务传递一次取消标志，然后给任务构造函数再传递一次。为什么需要提供取消标志两次呢?

答案是如果在任务实际启动前取消它，该任务的 TPL 基础设施有责任处理该取消操作，因为这些代码根本不会执行。通过得到的第一个任务的状态可以知道它被取消了。如果尝试对该任务调用 Start 方法，将会得到 InvalidOperationException 异常。

然后需要自己写代码来处理取消过程。这意味着我们对取消过程全权负责，并且在取消任务后，任务的状态仍然是 RanToCompletion，因为从 TPL 的视角来看，该任务正常完成了它的工作。辨别这两种情况是非常重要的，并且需要理解每种情况下职责的不同。

4.8　处理任务中的异常

本节将描述异步任务中处理异常这一重要的主题。我们将讨论任务中抛出异常的不同情

况，以及如何获取这些异常信息。

4.8.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter4\Recipe7 目录中。

4.8.2 实现方式

请执行以下步骤来了解如何在任务中处理异常：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
static int TaskMethod(string name, int seconds){
  Console.WriteLine("Task {0} is running on a thread id
    {1}. Is thread pool thread: {2}", name,
      Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread);
  Thread.Sleep(TimeSpan.FromSeconds(seconds));
  throw new Exception("Boom!");
  return 42 * seconds;
}
```

4. 在 Main 方法中加入以下代码片段：

```
Task<int> task;
try{
  task = Task.Run(() =>TaskMethod("Task 1", 2));
  int result = task.Result;
  Console.WriteLine("Result: {0}", result);
}
catch (Exception ex){
  Console.WriteLine("Exception caught: {0}", ex);
}
Console.WriteLine("----------------------------------------
  ------");
Console.WriteLine();

try{
  task = Task.Run(() =>TaskMethod("Task 2", 2));
  int result = task.GetAwaiter().GetResult();
  Console.WriteLine("Result: {0}", result);
```

```
}
catch (Exception ex){
  Console.WriteLine("Exception caught: {0}", ex);
}
Console.WriteLine("----------------------------------------
  ------");
Console.WriteLine();

var t1 = new Task<int>(() =>TaskMethod("Task 3", 3));
var t2 = new Task<int>(() =>TaskMethod("Task 4", 2));
var complexTask = Task.WhenAll(t1, t2);
var exceptionHandler = complexTask.ContinueWith(t =>
  Console.WriteLine("Exception caught: {0}", t.Exception),
    TaskContinuationOptions.OnlyOnFaulted);
t1.Start();
t2.Start();

Thread.Sleep(TimeSpan.FromSeconds(5));
```

5. 运行程序。

4.8.3　工作原理

当程序启动时，创建了一个任务并尝试同步获取任务结果。Result 属性的 Get 部分会使当前线程等待直到该任务完成，并将异常传播给当前线程。在这种情况下，通过 catch 代码块可以很容易地捕获异常，但是该异常是一个被封装的异常，叫做 AggregateException。在本例中，它里面包含一个异常，因为只有一个任务抛出了异常。可以访问 InnerException 属性来得到底层异常。

第二个例子与第一个非常相似，不同之处是使用 GetAwaiter 和 GetResult 方法来访问任务结果。这种情况下，无需封装异常，因为 TPL 基础设施会提取该异常。如果只有一个底层任务，那么一次只能获取一个原始异常，这种设计非常合适。

最后一个例子展示了两个任务抛出异常的情形。现在使用后续操作来处理异常。只有之前的任务完成前有异常时，该后续操作才会被执行。通过给后续操作传递 TaskContinuationOptions.OnlyOnFaulted 选项可以实现该行为。结果打印出了 AggregateException，其内部封装了两个任务抛出的异常。

4.8.4　更多信息

由于任务可以以非常不同的方式连接，因此结果的 AggregateException 异常可能包含其他内部包含普通异常的聚合异常。这些内部的聚合异常自身也可能包含其他的聚合异常。

为了摆脱这些对异常的封装，我们可以使用根聚合异常的 Flatten 方法。它将返回一个集合。该集合包含以层级结构中每个子聚合异常中的内部异常。

4.9 并行运行任务

本节展示了如何同时运行多个异步任务。我们将学习当所有任务都完成或任意一个任务完成了工作时，如何高效地得到通知。

4.9.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter4\Recipe8 目录中。

4.9.2 实现方式

请执行以下步骤来并行的运行任务：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
static int TaskMethod(string name, int seconds){
  Console.WriteLine("Task {0} is running on a thread id
    {1}. Is thread pool thread: {2}", name,
      Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread);
  Thread.Sleep(TimeSpan.FromSeconds(seconds));
  return 42 * seconds;
}
```

4. 在 Main 方法中加入以下代码片段：

```
var firstTask = new Task<int>(() =>TaskMethod("First Task",
  3));
var secondTask = new Task<int>(() =>TaskMethod("Second
  Task", 2));
var whenAllTask = Task.WhenAll(firstTask, secondTask);

whenAllTask.ContinueWith(t =>
  Console.WriteLine("The first answer is {0}, the second is
    {1}", t.Result[0], t.Result[1]),
      TaskContinuationOptions.OnlyOnRanToCompletion);

firstTask.Start();
secondTask.Start();
```

```
Thread.Sleep(TimeSpan.FromSeconds(4));

var tasks = new List<Task<int>>();
for (int i = 1; i< 4; i++)
{
  int counter = i;
  var task = new Task<int>(() =>TaskMethod(string.Format(
    "Task {0}", counter), counter));
  tasks.Add(task);
  task.Start();
}
while (tasks.Count> 0){
  var completedTask = Task.WhenAny(tasks).Result;
  tasks.Remove(completedTask);
  Console.WriteLine("A task has been completed with result
    {0}.", completedTask.Result);
}

Thread.Sleep(TimeSpan.FromSeconds(1));
```

5. 运行程序。

4.9.3　工作原理

当程序启动时，创建了两个任务。然后借助于 Task.WhenAll 方法，创建了第三个任务，该任务将会在所有任务完成后运行。该任务的结果提供了一个结果数组，第一个元素是第一个任务的结果，第二个元素是第二个任务的结果，以此类推。

然后我们创建了另外一系列任务，并使用 Task.WhenAny 方法等待这些任务中的任何一个完成。当有一个完成任务后，从列表中移除该任务并继续等待其他任务完成，直到列表为空。获取任务的完成进展情况或在运行任务时使用超时，都可以使用 Task.WhenAny 方法。例如，我们等待一组任务运行，并且使用其中一个任务用来记录是否超时。如果该任务先完成，则只需取消掉其他还未完成的任务。

4.10　使用 TaskScheduler 配置任务的执行

本节将描述处理任务的另一个重要方面，即通过异步代码与 UI 正确地交互。我们将学习什么是任务调度程序，为什么它如此重要，它如何损害应用程序，以及如何在使用它时避免错误。

4.10.1　准备工作

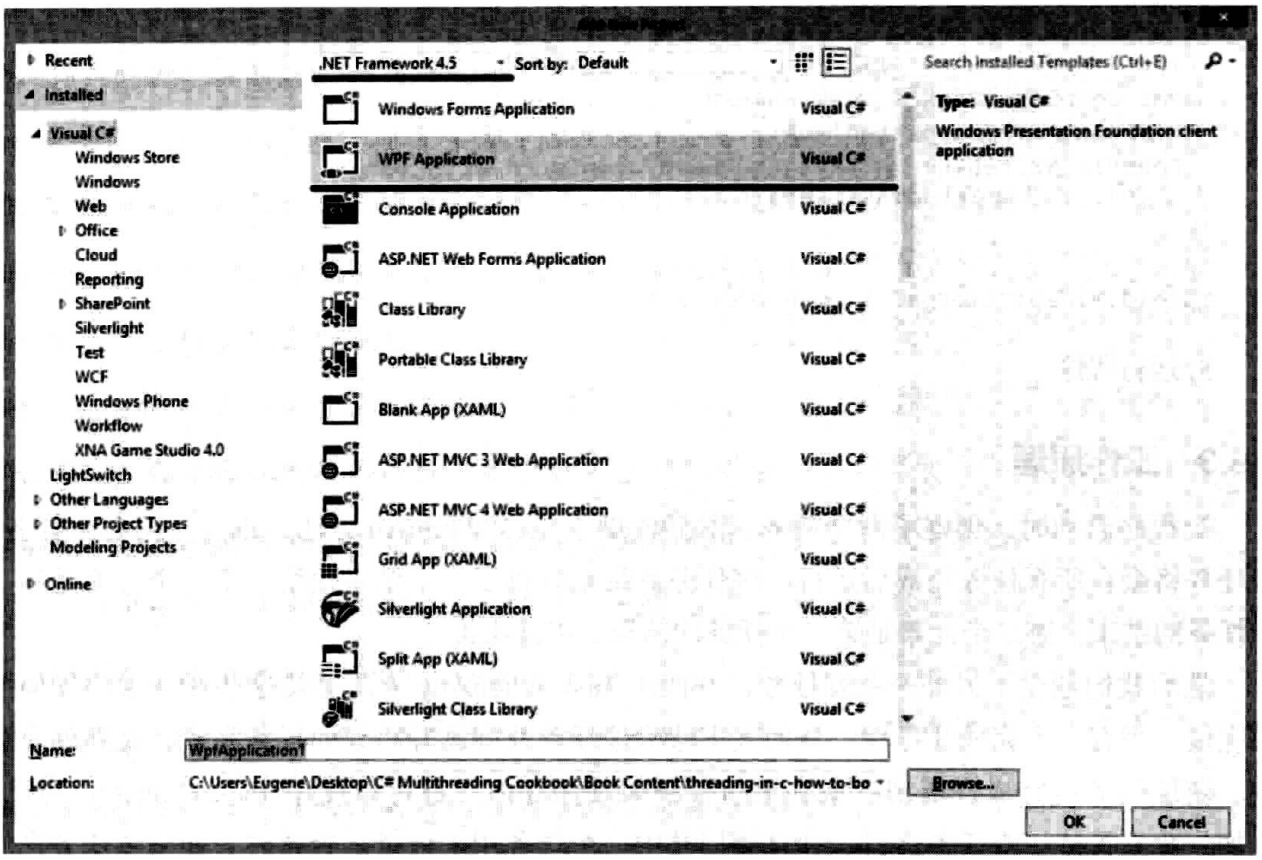
为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码

放置在 BookSamples\Chapter4\Recipe9 目录中。

4.10.2 实现方式

请执行以下步骤来使用 TaskScheduler 配置任务的执行：

1. 启动 Visual Studio 2012。新建一个 C# WPF 应用程序项目。这次我们需要一个有消息循环的 UI 线程，而命令行应用程序则没有。



2. 在 MainWindow.xaml 文件中，将下面的标记代码加入到一个网格元素中（即 <Grid> 和 </Grid> 标签间）：

```
<TextBlock Name="ContentTextBlock"
HorizontalAlignment="Left"
Margin="44,134,0,0"
VerticalAlignment="Top"
Width="425"
Height="40"/>
<Button Content="Sync"
HorizontalAlignment="Left"
Margin="45,190,0,0"
VerticalAlignment="Top"
Width="75"
```

```
Click="ButtonSync_Click"/>
<Button Content="Async"
HorizontalAlignment="Left"
Margin="165,190,0,0"
VerticalAlignment="Top"
Width="75"
Click="ButtonAsync_Click"/>
<Button Content="Async OK"

HorizontalAlignment="Left"

Margin="285,190,0,0"

VerticalAlignment="Top"

Width="75"

Click="ButtonAsyncOK_Click"/>
```

3. 在 MainWindow.xaml.cs 文件中使用以下 using 指令：

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Input;
```

4. 在 MainWindow 构造函数下面加入以下代码片段：

```
void ButtonSync_Click(object sender, RoutedEventArgs e){
  ContentTextBlock.Text = string.Empty;
  try {
    //string result = TaskMethod(TaskScheduler.
    //FromCurrentSynchronizationContext()).Result;
    string result = TaskMethod().Result;
    ContentTextBlock.Text = result;
  }
  catch (Exception ex) {
    ContentTextBlock.Text = ex.InnerException.Message;
  }
}

void ButtonAsync_Click(object sender, RoutedEventArgs e) {
  ContentTextBlock.Text = string.Empty;
  Mouse.OverrideCursor = Cursors.Wait;
  Task<string> task = TaskMethod();
  task.ContinueWith(t => {
    ContentTextBlock.Text = t.Exception.InnerException.
      Message;
    Mouse.OverrideCursor = null;
  },
  CancellationToken.None, TaskContinuationOptions.
    OnlyOnFaulted,
```

```
    TaskScheduler.FromCurrentSynchronizationContext());
}

void ButtonAsyncOK_Click(object sender, RoutedEventArgs e){
    ContentTextBlock.Text = string.Empty;
    Mouse.OverrideCursor = Cursors.Wait;
    Task<string> task = TaskMethod(TaskScheduler.
      FromCurrentSynchronizationContext());
    task.ContinueWith(t =>Mouse.OverrideCursor = null,
      CancellationToken.None,
      TaskContinuationOptions.None,
      TaskScheduler.FromCurrentSynchronizationContext());
}

Task<string> TaskMethod() {
    return TaskMethod(TaskScheduler.Default);
}

Task<string> TaskMethod(TaskScheduler scheduler) {
    Task delay = Task.Delay(TimeSpan.FromSeconds(5));

    return delay.ContinueWith(t => {
      string str = string.Format("Task is running on a thread
        id {0}. Is thread pool thread: {1}",
          Thread.CurrentThread.ManagedThreadId,
            Thread.CurrentThread.IsThreadPoolThread);
      ContentTextBlock.Text = str;
      return str;
    }, scheduler);
}
```

5. 运行程序。

4.10.3 工作原理

本例中引入了很多新鲜的东西。首先，创建了一个 WPF 应用程序，而不是一个命令行程序。这是很有必要的，因为我们需要一个拥有消息循环的用户界面线程来演示异步运行任务的不同情形。

TaskScheduler 是一个非常重要的抽象。该组件实际上负责如何执行任务。默认的任务调度程序将任务放置到线程池的工作线程中。这是非常常见的场景，所以 TPL 将其作为默认选项并不用奇怪。我们已经知道了如何同步运行任务，以及如何将任务附加到父任务上从而一起运行。现在让我们看看使用任务的其他方式。

当程序启动时，创建了一个包含三个按钮的窗口。第一个按钮调用了一个同步任务的执行。该代码被放置在 ButtonSync_Click 方法中。当任务运行时，我们甚至无法移动应用程序窗口。当用户界面线程忙于运行任务时，整个用户界面被完全冻结，在任务完成前无法响应

任何消息循环。对于 GUI 窗口程序来说这是一个相当不好的实践，我们需要找到一个方式来解决该问题。

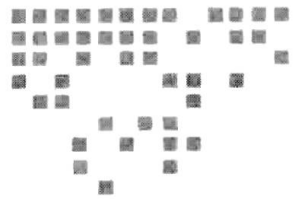
第二个问题是我们尝试从其他线程访问 UI 控制器。图形用户界面控制器从没有被设计为可被多线程使用，并且为了避免可能的错误，不允许从创建 UI 的线程之外的线程中访问 UI 组件。当我们尝试这样做时，得到了一个异常，该异常信息 5 秒后打印到了主窗口中。

为了解决第一个问题，我们尝试异步运行任务。第二个按钮就是这样做的。该代码被放置在 ButtonAsync_Click 方法中。当使用调试模式运行该任务时，将会看到该任务被放置在线程池中，最后将得到同样的异常。然而，当任务运行时用户界面一直保持响应。这是好事，但是我们仍需要除掉异常。

其实我们已经解决了该问题。给 TaskScheduler.FromCurrentSynchronizationContext 选项提供一个后续操作用于输出错误信息。如果不这样做，我们将无法看到错误信息，因为可能会得到在任务中产生的相同异常。该选项驱使 TPL 基础设施给 UI 线程的后续操作中放入代码，并借助 UI 线程消息循环来异步运行该代码。这解决了从其他线程访问 UI 控制器并仍保持 UI 处于响应状态的问题。

为了检查是否真的是这样，可以按下最后一个按钮来运行 ButtonAsyncOK_Click 方法中的代码。与其余例子不同之处在于我们将 UI 线程任务调度程序提供给了该任务。你将看到任务以异步的方式运行在 UI 线程中。UI 依然保持响应。甚至尽管等待光标处于激活状态，你仍可以按下另一个按钮。

然而使用 UI 线程运行任务有一些技巧。如果回到同步任务代码，取消对使用 UI 线程任务调度程序获取结果的代码行的注释，我们将永远得不到任何结果。这是一个经典的死锁情况：我们在 UI 线程队列中调度了一个操作，UI 线程等待该操作完成，但当等待时，它又无法运行该操作，这将永不会结束（甚至永不会开始）。如果在任务中调用 Wait 方法也会发生死锁。为了避免死锁，绝对不要通过任务调度程序在 UI 线程中使用同步操作，请使用 C# 5.0 中的 ContinueWith 或 async/await 方法。



Chapter 5 第 5 章

使用 C# 5.0

在本章中，我们将学习 C# 5.0 编程语言中的原生的异步编程方式。你将学到以下内容：

- 使用 await 操作符获取异步任务结果
- 在 lambda 表达式中使用 await 操作符
- 对连续的异步任务使用 await 操作符
- 对并行执行的异步任务使用 await 操作符
- 处理异步操作中的异常
- 避免使用捕获的同步上下文
- 使用 async void 方法
- 设计一个自定义的 awaitable 类型
- 对动态类型使用 await

5.1 简介

到现在为止，我们学习了任务并行库，这是微软提供的最新的异步编程基础设施。它允许我们以模块化的方式设计程序，来组合不同的异步操作。

遗憾的是，当阅读此类程序时仍然非常难理解程序的实际执行顺序。在大型程序中将会有许多相互依赖的任务和后续操作，用于运行其他后续操作的后续操作，处理异常的后续操作，并且它们都出现在程序代码中不同的地方。因此了解程序的先后执行次序变成了一个极具挑战性的问题。

另一个需要关注的问题是，能够接触用户界面控制器的每个异步任务是否得到了正确的同步上下文。程序只允许通过 UI 线程使用这些控制器，否则将会得到多线程访问异常。

说到异常，我们不得不使用单独的后续操作任务来处理在之前的异步操作中发生的错误。这又导致了分散在代码的不同部分的复杂的处理错误的代码，逻辑上无法相互关联。

为了解决这些问题，C# 5.0 的作者引入了新的语言特性，称为**异步函数**（asynchronous function）。它是 TPL 之上的更高级别的抽象，真正简化了异步编程。正如在第 4 章提到的，抽象隐藏了主要的实现细节，使得程序员无须考虑许多重要的事情，从而使异步编程更容易。了解异步函数背后的概念是非常重要的，有助于我们编写健壮的高扩展性的应用程序。

要创建一个异步函数，首先需要用 async 关键字标注一个方法。如果不先做这个，就不可能拥有 async 属性或事件访问方法和构造函数。代码如下所示：

```
async Task<string> GetStringAsync()
{
  await Task.Delay(TimeSpan.FromSeconds(2));
  return "Hello, World!";
}
```

另一个重要的事实是，异步函数必须返回 Task 或 Task<T> 类型。可以使用 async void 方法，但是更推荐使用 async Task 方法。使用 async void 方法唯一合理的地方是在程序中使用顶层 UI 控制器事件处理器的时候。

使用 async 关键字标注的方法内部，可以使用 await 操作符。该操作符可与 TPL 的任务一起工作，并获取该任务中异步操作的结果。在本章中稍后会讲述细节。在 async 方法外不能使用 await 关键字，否则会有编译错误。另外，异步函数在其代码中至少要拥有一个 await 操作符。然而，如果没有只会导致编译警告，而不是编译错误。

需要注意的是，在执行完 await 调用的代码行后该方法会立即返回。如果是同步执行，执行线程将会阻塞两秒然后返回结果。这里当执行完 await 操作后，立即将工作线程放回线程池的过程中，我们会异步等待。2 秒后，我们又一次从线程池中得到工作线程并继续运行其中剩余的异步方法。这允许我们在等待 2 秒时重用工作线程做些其他事，这对提高应用程序的可伸缩性非常重要。借助于异步函数我们拥有了线性的程序控制流，但它的执行依然是异步的。这虽然好用，但是难以理解。本章将帮助你学习异步函数所有重要的方面。



以我的自身经验而言，如果程序中有两个连续的 await 操作符，此时程序如何工作有一个常见的误解。很多人认为如果在另一个异步操作之后使用 await 函数，它们将会并行运行。然而，事实上它们是顺序运行的，即第一个完成后第二个才会开始运行。记住这一点很重要，在本章中稍后会覆盖该细节。

在 C# 5.0 中关联 async 和 await 有一定的限制。例如，不能把控制台程序的 Main 方法标记为 async。不能在 catch、finally、lock 或 unsafe 代码块中使用 await 操作符。不允许对任何异步函数使用 ref 或 out 参数。还有其他微妙的地方，但是以上已经包括了主要的需要注意的地方。

异步函数会被 C# 编译器在后台编译成复杂的程序结构。这里我不会说明该细节。生成的代码与另一个 C# 构造很类似，称为**迭代器**。生成的代码被实现为一种状态机。尽管很多程序员几乎开始为每个方法使用 async 修饰符，我还是想强调如果方法本来无需异步或并行运行，那么将该方法标注为 async 是没有道理的。调用 async 方法会有显著的性能损失，通常的方法调用比使用 async 关键字的同样的方法调用要快上 40 ~ 50 倍。请注意这一点。

在本章中我们将学习如何使用 C# 5.0 中的 async 和 await 关键字实现异步操作。本章将讲述如何使用 await 按顺序或并行地执行异步操作，还将讨论如何在 lambda 表达式中使用 await，如何处理异常，以及在使用 async void 方法时如何避免陷阱。在本章结束前，我们会深入探究同步上下文传播机制并学习如何创建自定义的 awaitable 对象，从而无需使用任务。

5.2 使用 await 操作符获取异步任务结果

本节将讲述使用异步函数的基本场景。我们将比较使用 TPL 和使用 await 操作符获取异步操作结果的不同之处。

5.2.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter5\Recipe1 目录中。

5.2.2 实现方式

请执行以下步骤来使用 await 操作符获取异步任务结果：

1. 启动 Visual Studio 2012。新建一个 C# 控制台应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
static Task AsynchronyWithTPL()
{
  Task<string> t = GetInfoAsync("Task 1");
  Task t2 = t.ContinueWith(task => Console.WriteLine(
    t.Result), TaskContinuationOptions.NotOnFaulted);
  Task t3 = t.ContinueWith(task => Console.WriteLine(
    t.Exception.InnerException), TaskContinuationOptions.
      OnlyOnFaulted);

  return Task.WhenAny(t2, t3);
}

async static Task AsynchronyWithAwait()
{
  try
  {
    string result = await GetInfoAsync("Task 2");
    Console.WriteLine(result);
  }

  catch (Exception ex)
  {
    Console.WriteLine(ex);
  }
}

async static Task<string> GetInfoAsync(string name)
{
  await Task.Delay(TimeSpan.FromSeconds(2));
  //throw new Exception("Boom!");

  return string.Format("Task {0} is running on a thread id
    {1}. Is thread pool thread: {2}", name,
      Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread);
}
```

4. 在 Main 方法中加入以下代码片段：

```
Task t = AsynchronyWithTPL();
t.Wait();

t = AsynchronyWithAwait();
t.Wait();
```

5. 运行程序。

5.2.3 工作原理

当程序运行时运行了两个异步操作。其中一个是标准的 TPL 模式的代码，第二个使用了 C# 的新特性 async 和 await。AsynchronyWithTPL 方法启动了一个任务，运行两秒后返回关

于工作线程信息的字符串。然后我们定义了一个后续操作，用于在异步操作完成后打印出该操作结果，还有另一个后续操作，用于万一有错误发生时打印出异常的细节。最终，返回了一个代表其中一个后续操作任务的任务，并等待其在 Main 函数中完成。

在 AsynchronyWithAwait 方法中，我们对任务使用 await 并得到了相同的结果。这和编写通常的同步代码的风格一样，即我们获取任务的结果，打印出结果，如果任务完成时带有错误则捕获异常。关键不同的是这实际上是一个异步程序。使用 await 后，C# 立即创建了一个任务，其有一个后续操作任务，包含了 await 操作符后面的所有剩余代码。这个新任务也处理了异常传播。然后，将该任务返回到主方法中并等待其完成。



请注意根据底层异步操作的性质和当前异步的上下文，执行异步代码的具体方式可能会不同。稍后在本章中会解释这一点。

因此可以看到程序的第一部分和第二部分在概念上是等同的，但是在第二部分中 C# 编译器隐式地处理了异步代码。事实上，第二部分比第一部分更复杂，接下来我们将讲述细节。

请记住在 Windows GUI 或 ASP.NET 之类的环境中不推荐使用 Task.Wait 和 Task.Result 方法。如果程序员不是百分百地清楚代码在做什么，很可能会导致死锁。在第 4 章的 4.10 节中，在 WPF 应用程序中使用 Task.Result 时已经演示了该一点。

请取消对 GetInfoAsync 方法的 throw new Exception 代码行的注释来测试异常处理是否工作。

5.3 在 lambda 表达式中使用 await 操作符

本节将展示如何在 lambda 表达式中使用 await。我们将编写一个使用了 await 的匿名方法，并且获取异步执行该方法的结果。

5.3.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter5\Recipe2 目录中。

5.3.2 实现方式

请执行以下步骤来编写一个使用了 await 的匿名方法，并得到在 lambada 表达式中使用 await 操作符异步执行该方法后产生的结果：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
async static Task AsynchronousProcessing()
{
  Func<string, Task<string>> asyncLambda = async name => {
    await Task.Delay(TimeSpan.FromSeconds(2));
    return string.Format("Task {0} is running on a thread
      id {1}. Is thread pool thread: {2}", name,
        Thread.CurrentThread.ManagedThreadId,
          Thread.CurrentThread.IsThreadPoolThread);
  };

  string result = await asyncLambda("async lambda");

  Console.WriteLine(result);
}
```

4. 在 Main 方法中加入以下代码片段：

```
Task t = AsynchronousProcessing();
t.Wait();
```

5. 运行程序。

5.3.3　工作原理

首先，由于不能在 Main 方法中使用 async，我们将异步函数移到了 AsynchronousProcessing 方法中。然后使用 async 关键字声明了一个 lambda 表达式。由于任何 lambda 表达式的类型都不能通过 lambda 自身来推断，所以不得不显式向 C# 编译器指定它的类型。在本例中，该类型说明该 lambda 表达式接受一个字符串参数，并返回一个 Task<string> 对象。

接着，我们定义了 lambda 表达式体。有个问题是该方法被定义为返回一个 Task<string> 对象，但实际上返回的是字符串，却没有编译错误！这是因为 C# 编译器自动产生一个任务并返回给我们。

最后一步是等待异步 lambda 表达式执行并打印出结果。

5.4　对连续的异步任务使用 await 操作符

本节将展示当代码中有多个连续的 await 方法时程序的实际流程是怎样的。我们将学习

如何阅读有 await 方法的代码，以及理解为什么 await 调用是异步操作。

5.4.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter5\Recipe3 目录中。

5.4.2 实现方式

请执行以下步骤来了解连续的 await 方法出现时程序的执行流程：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
static Task AsynchronyWithTPL()
{
  var containerTask = new Task(() => {
    Task<string> t = GetInfoAsync("TPL 1");
    t.ContinueWith(task => {
      Console.WriteLine(t.Result);
      Task<string> t2 = GetInfoAsync("TPL 2");
      t2.ContinueWith(innerTask =>
        Console.WriteLine(innerTask.Result),
          TaskContinuationOptions.NotOnFaulted |
            TaskContinuationOptions.AttachedToParent);
      t2.ContinueWith(innerTask =>
        Console.WriteLine(innerTask.Exception.
          InnerException),
           TaskContinuationOptions.OnlyOnFaulted |
             TaskContinuationOptions.AttachedToParent);
      },
      TaskContinuationOptions.NotOnFaulted |
        TaskContinuationOptions.AttachedToParent);

    t.ContinueWith(task =>
      Console.WriteLine(t.Exception.InnerException),
        TaskContinuationOptions.OnlyOnFaulted |
          TaskContinuationOptions.AttachedToParent);
  });

  containerTask.Start();
  return containerTask;
}
```

```
async static Task AsynchronyWithAwait()
{
  try
  {
    string result = await GetInfoAsync("Async 1");
    Console.WriteLine(result);
    result = await GetInfoAsync("Async 2");
    Console.WriteLine(result);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex);
  }
}

async static Task<string> GetInfoAsync(string name)
{
  Console.WriteLine("Task {0} started!", name);
  await Task.Delay(TimeSpan.FromSeconds(2));
  if(name == "TPL 2")
    throw new Exception("Boom!");
  return string.Format("Task {0} is running on a thread id
    {1}. Is thread pool thread: {2}",
      name, Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread);
}
```

4. 在 Main 方法中加入以下代码片段：

```
Task t = AsynchronyWithTPL();
t.Wait();

t = AsynchronyWithAwait();
t.Wait();
```

5. 运行程序。

5.4.3 工作原理

当程序运行时，与上节一样运行了两个异步操作。然而这次从 AsynchronyWithAwait 方法讲起。它看起来仍然像平常的同步代码，唯一不同之处是使用了两个 await 声明。最重要的一点是该代码依然是顺序执行的，Async 2 任务只有等之前的任务完成后才会开始执行。当阅读该代码时，程序流很清晰，可以看到什么先运行，什么后运行。但该程序如何是异步程序呢？首先，它不总是异步的。当使用 await 时如果一个任务已经完成，我们会异步地得到该任务结果。否则，当在代码中看到 await 声明时，通常的行为是方法执行到该 await 代码行时将立即返回，并且剩下的代码将会在一个后续操作任务中运行。因此等待操作结果时并没有阻塞程序执行，这是一个异步调用。当 AsynchronyWithAwait 方法中

的代码在执行时，除了在 Main 方法中调用 t.Wait 外，我们可以执行任何其他任务。然而，主线程必须等待直到所有异步操作完成，否则主线程完成后所有运行异步操作的后台线程会停止运行。

AsynchronyWithTPL 方法模仿了 AsynchronyWithAwait 的程序流。我们需要一个容器任务来处理所有相互依赖的任务。然后启动主任务，给其加了一组后续操作。当该任务完成后，会打印出其结果。然后又启动了一个任务，在该任务完成后会依次运行更多的后续操作。为了测试对异常的处理，当运行第二个任务时故意抛出一个异常，并打印出异常信息。这组后续操作创建了与第一个方法中一样的程序流。如果用它与 await 方法比较，可以看到它更容易阅读和理解。唯一的技巧是请记住异步并不总是意味着并行执行。

5.5 对并行执行的异步任务使用 await 操作符

本节将学习如何使用 await 来并行地运行异步任务，而不是采用常用的顺序执行。

5.5.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter5\Recipe4 目录中。

5.5.2 实现方式

请执行以下步骤以了解如何使用 await 操作符来并行地执行任务：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
async static Task AsynchronousProcessing()
{
  Task<string> t1 = GetInfoAsync("Task 1", 3);
  Task<string> t2 = GetInfoAsync("Task 2", 5);

  string[] results = await Task.WhenAll(t1, t2);
  foreach (string result in results)
  {
    Console.WriteLine(result);
  }
```

```
}

async static Task<string> GetInfoAsync(string name, int seconds)
{
  await Task.Delay(TimeSpan.FromSeconds(seconds));
  /*await Task.Run(() =>
    Thread.Sleep(TimeSpan.FromSeconds(seconds)));*/
  return string.Format("Task {0} is running on a thread id
    {1}. Is thread pool thread: {2}", name,
      Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread);
}
```

4. 在 Main 方法中加入以下代码片段：

```
Task t = AsynchronousProcessing();
t.Wait();
```

5. 运行程序。

5.5.3　工作原理

这里定义了两个异步任务，分别运行 3 秒和 5 秒。然后使用 Task.WhenAll 辅助方法创建了另一个任务，该任务只有在所有底层任务完成后才会运行。之后我们等待该组合任务的结果。5 秒后，我们获取了所有结果，说明了这些任务是同时运行的。

然而这里观察到一个有意思的现象。当运行该程序时，你可能注意到这两个任务似乎是被线程池中的同一个工作线程执行的。当我们并行运行任务时怎么可能发生这样的事情呢？为了让事情更有趣，我们来注释掉 GetIntroAsync 方法中的 await Task.Delay 代码行，并解除对 await Task.Run 代码行的注释，然后再次运行程序。

我们会看到该情况下两个任务会被不同的工作线程执行。不同之处是 Task.Delay 在幕后使用了一个计时器，过程如下：从线程池中获取工作线程，它将等待 Task.Delay 方法返回结果。然后，Task.Delay 方法启动计时器并指定一块代码，该代码会在计时器时间到了 Task.Delay 方法中指定的秒数后被调用。之后立即将工作线程返回到线程池中。当计时器事件运行时，我们又从线程池中任意获取一个可用的工作线程（可能就是运行一个任务时使用的线程）并运行计时器提供给它的代码。

当使用 Task.Run 方法时，从线程池中获取了一个工作线程并将其阻塞几秒，具体秒数由 Thread.Sleep 方法提供。然后获取了第二个工作线程并且也将其阻塞。在这种场景下，我们消费了两个工作线程，而它们绝对什么事没做，因为在它们等待时不能执行任何其他操作。

我们将在第 9 章中讨论第一个场景的细节。在第 9 章我们将讨论用大量的异步操作进行数据输入和输出。尽可能地使用第一种方式是创建高伸缩性的服务器程序的关键。

5.6 处理异步操作中的异常

本节将描述在 C# 中使用异步函数时如何处理异常。我们将学习对多个并行的异步操作使用 await 时如何聚合异常。

5.6.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 ookSamples\Chapter5\Recipe5 目录中。

5.6.2 实现方式

请执行以下步骤来了解如何在异步操作中处理异常：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
async static Task AsynchronousProcessing()
{
  Console.WriteLine("1. Single exception");

  try
  {
    string result = await GetInfoAsync("Task 1", 2);
    Console.WriteLine(result);
  }
  catch (Exception ex)
  {
    Console.WriteLine("Exception details: {0}", ex);
  }

  Console.WriteLine();
  Console.WriteLine("2. Multiple exceptions");

  Task<string> t1 = GetInfoAsync("Task 1", 3);
  Task<string> t2 = GetInfoAsync("Task 2", 2);
  try
  {
    string[] results = await Task.WhenAll(t1, t2);
    Console.WriteLine(results.Length);
  }
  catch (Exception ex)
  {
    Console.WriteLine("Exception details: {0}", ex);
  }
```

```
    Console.WriteLine();
    Console.WriteLine("2. Multiple exceptions with
AggregateException");

    t1 = GetInfoAsync("Task 1", 3);
    t2 = GetInfoAsync("Task 2", 2);
    Task<string[]> t3 = Task.WhenAll(t1, t2);
    try
    {
      string[] results = await t3;
      Console.WriteLine(results.Length);
    }
    catch
    {
      var ae = t3.Exception.Flatten();
      var exceptions = ae.InnerExceptions;
      Console.WriteLine("Exceptions caught: {0}", exceptions.Count);
      foreach (var e in exceptions)
      {
        Console.WriteLine("Exception details: {0}", e);
        Console.WriteLine();
      }
    }
}

async static Task<string> GetInfoAsync(string name, int seconds)
{
  await Task.Delay(TimeSpan.FromSeconds(seconds));
  throw new Exception(string.Format("Boom from {0}!", name));
}
```

4. 在 Main 方法中加入以下代码片段：

```
Task t = AsynchronousProcessing();
t.Wait();
```

5. 运行程序。

5.6.3 工作原理

我们运行了三个场景来展示在 C# 中使用 async 和 await 时关于错误处理的最常见情况。第一种情况是最简单的，并且与常见的同步代码几乎完全一样。我们只使用 try/catch 声明即可获取异常细节。

一个很常见的错误是对一个以上的异步操作使用 await 时还使用以上方式。如果仍像第一种情况一样使用 catch 代码块，则只能从底层的 AggregateException 对象中得到第一个异常。

为了收集所有异常信息，可以使用 await 任务的 Exception 属性。在第三种情况中，我们使用 AggregateException 的 Flatten 方法将层级异常放入一个列表，并且从中提取出所有的底层异常。

5.7 避免使用捕获的同步上下文

本节描述了当使用 await 来获取异步操作结果时，同步上下文行为的细节。我们将学习如何以及何时关闭同步上下文流。

5.7.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter5\Recipe6 目录中。

5.7.2 实现方式

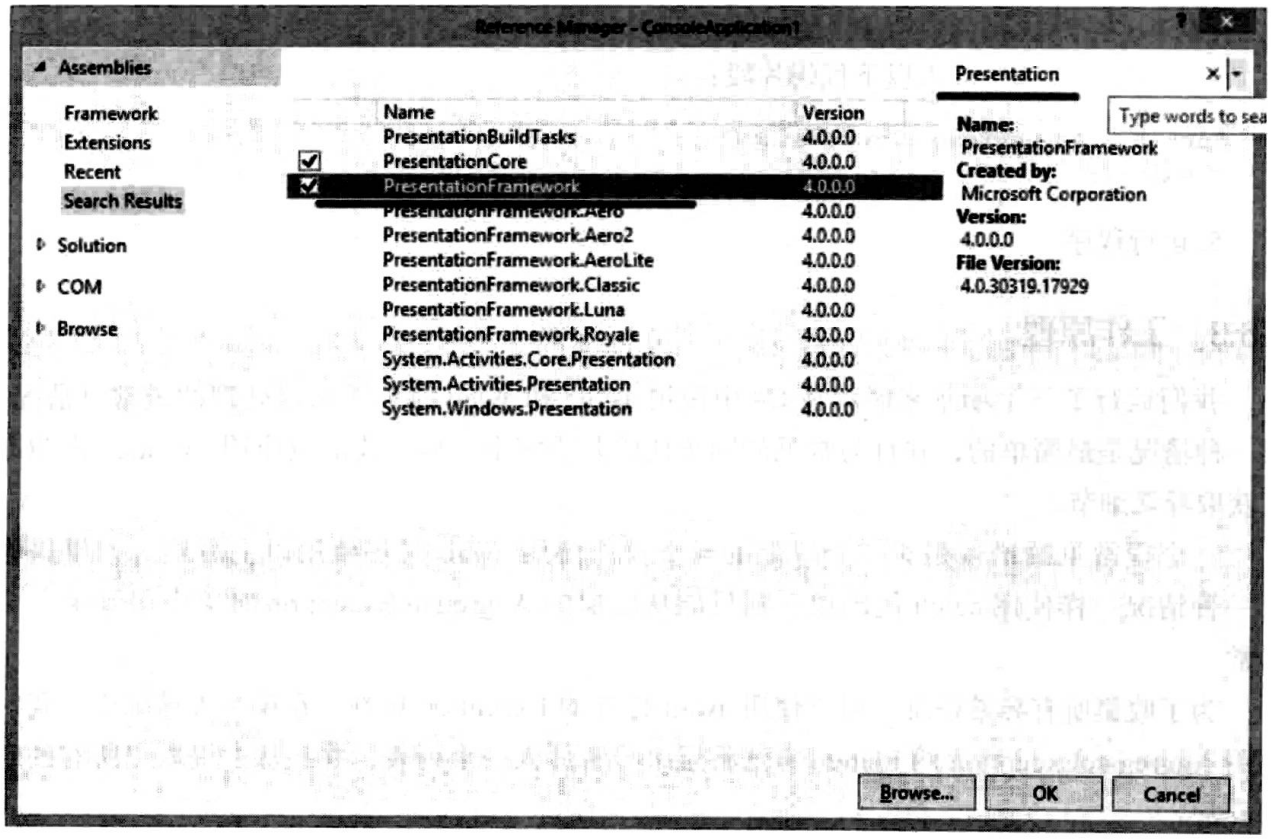
请执行以下步骤来了解当使用 await 时同步上下文行为的细节，以及如何及何时关闭同步上下文流：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 加入对 Windows Presentation Foundation 库的引用。

（1）右键点击项目中的**引用**文件夹，选择**添加引用…**菜单选项。

（2）添加对 PresentationCore、PresentationFramework、System.Xaml 及 Windows.Base 库的引用。你可以使用引用管理对话框的搜索功能，如下图所示：



3. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Diagnostics;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
```

4. 在 Main 方法下面加入以下代码片段：

```
private static Label _label;

async static void Click(object sender, EventArgs e)
{
  _label.Content = new TextBlock {Text = "Calculating..."};
  TimeSpan resultWithContext = await Test();
  TimeSpan resultNoContext = await TestNoContext();
  /*TimeSpan resultNoContext = await
    TestNoContext().ConfigureAwait(false);*/
  var sb = new StringBuilder();
  sb.AppendLine(string.Format("With the context: {0}",
    resultWithContext));
  sb.AppendLine(string.Format("Without the context: {0}",
    resultNoContext));
  sb.AppendLine(string.Format("Ratio: {0:0.00}",
    resultWithContext.TotalMilliseconds/resultNoContext.
      TotalMilliseconds));
  _label.Content = new TextBlock {Text = sb.ToString()};
}

async static Task<TimeSpan> Test()
{
  const int iterationsNumber = 100000;
  var sw = new Stopwatch();
  sw.Start();
  for (int i = 0; i < iterationsNumber; i++)
  {
    var t = Task.Run(() => { });
    await t;
  }
  sw.Stop();
  return sw.Elapsed;
}

async static Task<TimeSpan> TestNoContext()
{
  const int iterationsNumber = 100000;
  var sw = new Stopwatch();
  sw.Start();
  for (int i = 0; i < iterationsNumber; i++)
  {
    var t = Task.Run(() => { });
    await t.ConfigureAwait(
      continueOnCapturedContext: false);
  }
```

```
  sw.Stop();
  return sw.Elapsed;
}
```

5. 将 Main 方法替换为如下代码片段：

```
[STAThread]
static void Main(string[] args)
{
  var app = new Application();
  var win = new Window();
  var panel = new StackPanel();
  var button = new Button();
  _label = new Label();
  _label.FontSize = 32;
  _label.Height = 200;
  button.Height = 100;
  button.FontSize = 32;
  button.Content = new TextBlock {
    Text = "Start asynchronous operations"};
  button.Click += Click;
  panel.Children.Add(_label);
  panel.Children.Add(button);
  win.Content = panel;
  app.Run(win);

  Console.ReadLine();
}
```

6. 运行程序。

5.7.3 工作原理

在本例中，我们将学习异步函数默认行为的最重要的方面之一。我们已经从第 4 章中了解了任务调度程序和同步上下文。默认情况下，await 操作符会尝试捕获同步上下文，并在其中执行代码。我们已经知道这有助于我们编写与用户界面控制器协作的异步代码。另外，使用 await 不会发生在之前章节中描述过的死锁情况，因为当等待结果时并不会阻塞 UI 线程。

这是合理的，但是让我们看看潜在会发生什么事。在本例中，我们使用编程方式创建了一个 Windows Presentation Foundation 应用程序并订阅了它的按钮点击事件。当点击该按钮时，运行了两个异步操作。其中一个使用了一个常规的 await 操作符，另一个使用了带 false 参数值的 ConfigureAwait 方法。false 参数明确指出我们不能对其使用捕获的同步上下文来运行后续操作代码。在每个操作中，我们测量了执行完成花费的时间，然后将各自的时间和比例显示在主屏幕上。

结果看到常规的 await 操作符花费了更多的时间来完成。这是因为我们向 UI 线程中放

入了成百上千个后续操作任务，这会使用它的消息循环来异步地执行这些任务。在本例中，我们无需在 UI 线程中运行该代码，因为异步操作并未访问 UI 组件。使用带 false 参数值的 ConfigureAwait 方法是一个更高效的方案。

还有一件事值得一提。尝试运行程序并只点击按钮然后等待结果，然后再这样做一次，但是这次点击按钮后尝试随机地拖拽应用程序窗口从一侧到另一侧。你将注意到在捕获的同步上下文中的代码执行速度变慢了！这个有趣的副作用完美演示了异步编程是多么危险。经历类似的情况是非常容易的，而且如果你之前从未经历过这样的情况，那么几乎不可能通过调试来找出问题所在。

公平起见，让我们来看看相反的情况。在前面的代码片段中，在 Click 方法中，取消注释的代码行，并注释掉紧挨着它的前一行代码。当运行程序时，我们将得到多线程控制访问异常，因为设置 Label 控制器文本的代码不会放置到捕捉的上下文中，而是在线程池的工作线程中执行。

5.8　使用 async void 方法

本节描述了为什么使用 async void 方法非常危险。我们将学习在哪种情况下可使用该方法，以及如何尽可能地替代该方法。

5.8.1　准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter5\Recipe7 目录中。

5.8.2　实现方式

请执行以下步骤来学习如何使用 async void 方法：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
async static Task AsyncTaskWithErrors()
{
  string result = await GetInfoAsync("AsyncTaskException",
    2);
```

```
  Console.WriteLine(result);
}

async static void AsyncVoidWithErrors()
{
  string result = await GetInfoAsync("AsyncVoidException",
    2);
  Console.WriteLine(result);
}

async static Task AsyncTask()
{
  string result = await GetInfoAsync("AsyncTask", 2);
  Console.WriteLine(result);
}

private static async void AsyncVoid()
{
  string result = await GetInfoAsync("AsyncVoid", 2);
  Console.WriteLine(result);
}

async static Task<string> GetInfoAsync(string name,
  int seconds)
{
  await Task.Delay(TimeSpan.FromSeconds(seconds));
  if(name.Contains("Exception"))
    throw new Exception(string.Format("Boom from {0}!",
      name));
  return string.Format("Task {0} is running on a thread id
    {1}. Is thread pool thread: {2}", name,
      Thread.CurrentThread.ManagedThreadId,
        Thread.CurrentThread.IsThreadPoolThread);
}
```

4. 在 Main 方法中加入以下代码片段：

```
Task t = AsyncTask();
t.Wait();

AsyncVoid();
Thread.Sleep(TimeSpan.FromSeconds(3));

t = AsyncTaskWithErrors();
while(!t.IsFaulted)
{
  Thread.Sleep(TimeSpan.FromSeconds(1));
}
Console.WriteLine(t.Exception);

//try
```

```
//{
//  AsyncVoidWithErrors();
//  Thread.Sleep(TimeSpan.FromSeconds(3));
//}
//catch (Exception ex)
//{
//  Console.WriteLine(ex);
//}

//int[] numbers = new[] {1, 2, 3, 4, 5};
//Array.ForEach(numbers, async number => {
//  await Task.Delay(TimeSpan.FromSeconds(1));
//  if (number == 3) throw new Exception("Boom!");
//  Console.WriteLine(number);
//});

Console.ReadLine();
```

5. 运行程序。

5.8.3 工作原理

当程序启动时，我们通过调用 AsyncTask 和 AsyncVoid 这两个方法启动了两个异步操作。第一个方法返回一个 Task 对象，而另一个由于被声明为 async void 所以没有返回值。由于它们都是异步的所以都会立即返回。但是第一个方法通过返回的任务状态或对其调用 Wait 方法从而很容易实现监控。等待第二个方法完成的唯一方式是确切地等待多长时间，因为我们没有声明任何对象可以监控该异步操作的状态。当然可以使用某种共享的状态变量，将其设置到 async void 方法中，并从调用方法中检查其值，但返回一个 Task 对象的方式更好些。

最危险的部分是异常处理。使用 async void 方法，异常处理方法将被放置到当前的同步上下文中，在本例中即线程池中。线程池中未被处理的异常会终结整个进程。使用 AppDomain.UnhandledException 事件可以拦截未被处理的异常，但不能从拦截的地方恢复进程。为了重现该场景，可以取消 Main 方法中对 try/catch 代码块的注释，然后运行程序。

关于使用 async void lambda 表达式的另一个事实是：它们与 Action 类型是兼容的，而 Action 类型在标准 .NET Framework 类库中的使用非常广泛。在 lambda 表达式中很容易忘记对异常的处理，这将再次导致程序崩溃。可以取消在 Main 方法中第二个被注释的代码块的注释来重现该场景。

强烈建议只在 UI 事件处理器中使用 async void 方法。在其他所有的情况下，请使用返回 Task 的方法。

5.9 设计一个自定义的 awaitable 类型

本节将展示如何设计一个与 await 操作符兼容的非常基础的 awaitable 类型。

5.9.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter5\Recipe8 目录中。

5.9.2 实现方式

请执行以下步骤来设计一个自定义的 awaitable 类型：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Runtime.CompilerServices;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
async static Task AsynchronousProcessing()
{
  var sync = new CustomAwaitable(true);
  string result = await sync;
  Console.WriteLine(result);

  var async = new CustomAwaitable(false);
  result = await async;

  Console.WriteLine(result);
}

class CustomAwaitable
{
  public CustomAwaitable(bool completeSynchronously)
  {
    _completeSynchronously = completeSynchronously;
  }

  public CustomAwaiter GetAwaiter()
  {
    return new CustomAwaiter(_completeSynchronously);
  }

  private readonly bool _completeSynchronously;
}

class CustomAwaiter : INotifyCompletion
```

```
{
  private string _result = "Completed synchronously";
  private readonly bool _completeSynchronously;

  public bool IsCompleted { get {
    return _completeSynchronously; } }

  public CustomAwaiter(bool completeSynchronously)
  {
    _completeSynchronously = completeSynchronously;
  }

  public string GetResult()
  {
    return _result;
  }

  public void OnCompleted(Action continuation)
  {
    ThreadPool.QueueUserWorkItem( state => {
      Thread.Sleep(TimeSpan.FromSeconds(1));
      _result = GetInfo();
      if(continuation != null) continuation();
    });
  }

  private string GetInfo()
  {
    return string.Format("Task is running on a thread id
      {0}. Is thread pool thread: {1}", name,
        Thread.CurrentThread.ManagedThreadId,
          Thread.CurrentThread.IsThreadPoolThread);
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
Task t = AsynchronousProcessing();
t.Wait();
```

5. 运行程序。

5.9.3 工作原理

为了与 await 操作符保持兼容，类型应当遵守在 C# 5.0 规格说明中的规定的一些要求。如果你安装了 Visual Studio 2012，那么可以在 C:\Program Files\Microsoft Visual Studio 11.0\VC#\Specifications\1033（假设你使用的是默认安装路径）目录中找到该规格说明文档。

在规格说明文档的 7.7.7.1 节，我们发现了 awaitable 表达式的定义：

Await 表达式的任务被要求是 awaitable。如果一个表达式 t 满足下面任意一条则认为是 awaitable 的：

- t 是动态编译时的类型
- t 有一个名为 GetAwaiter 的可访问的实例或扩展方法。该方法没有参数和类型参数，并且返回值类型 A 满足以下所有条件：
 - A 实现了 System.Runtime.CompilerServices.INotifyCompletion 接口（为简单起见，以后简称为 INotifyCompletion）。
 - A 有一个可访问的、可读的类型为 bool 的实例属性 IsCompleted。
 - A 有一个名为 GetResult 的可访问的实例方法，该方法没有任何参数和类型参数。

这些信息足够我们开始了。首先我们定义一个 awaitable 类型 CustomAwaitable，并实现 GetAwaiter 方法，该方法返回一个 CustomAwaiter 类型的实例。CustomAwaiter 实现了 INotifyCompletion 接口，拥有类型为 bool 的 IsCompleted 属性，并且有 GetResult 方法，该方法返回一个字符串类型。最后，我们写了一些代码来创建两个 CustomAwaitable 对象并对其使用 await 关键字。

现在我们应该理解 await 表达式执行的方式了。这里并没有引用规格说明文档，以免陷入不必要的细节。基本上，如果 IsCompleted 属性返回 true，则只需同步调用 GetResult 方法。这种做法防止了该操作已经完成后我们仍然为执行异步任务而分配资源。通过给 CustomAwaitable 对象的构造函数传递 completeSynchronously 参数来展示该场景。

另外，我们给 CustomAwaiter 的 OnCompleted 方法注册了一个回调函数并启动该异步操作。当操作完成时，就会调用提供的回调函数，该回调函数将会通过调用 CustomAwaiter 对象的 GetResult 方法来获取结果。



该实现只适用于教学目的。当你编写异步函数时，最自然的方式还是使用标准的 Task 类型。只有具有无法使用 Task 的坚实理由，你才可以自定义 awaitable 类型，并且你需要真的知道你在做什么。

还有很多与设计自定义的 awaitable 类型相关的主题，例如 ICriticalNotifyCompletion 接口的实现及同步上下文传播。在明白了如何设计一个 awaitable 类型的基础知识后，你将能够使用 C# 语言规格说明和其他的信息源来毫不费力地找出你需要的细节。但我还是要强调最好还是使用 Task 类型，除非你有一个非常好的理由不这样做。

5.10 对动态类型使用 await

本节展示了如何设计一个非常基本的类型，该类型能够与 await 操作符和动态 C# 类型兼容。

5.10.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。还需要从网上下载 NuGet 包。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter5\Recipe9 目录中。

5.10.2 实现方式

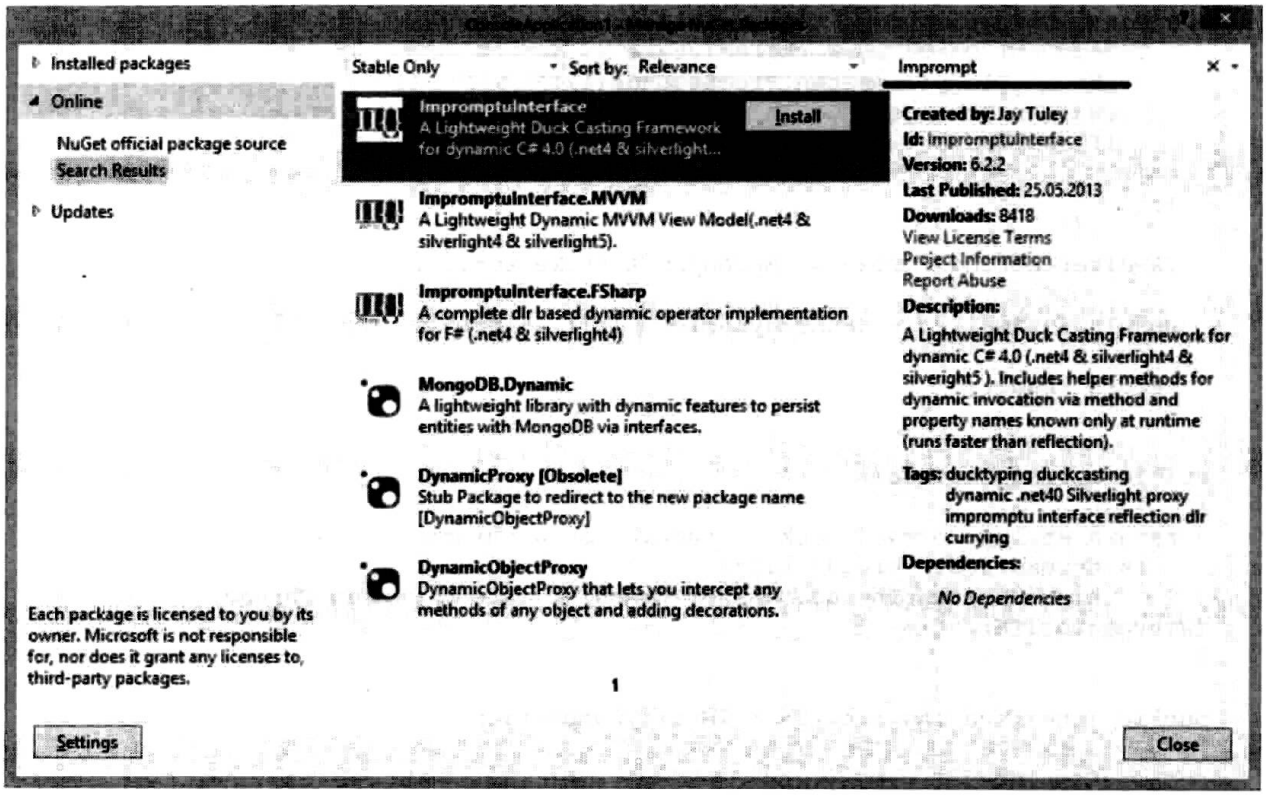
请执行以下步骤来学习如何对动态类型使用 await：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 请执行以下步骤来添加对 ImpromptuInterface NuGet 包的引用：

（1）右键点击项目中的**引用**文件夹，并选择管理 NuGet 包……菜单选项。

（2）添加对你喜欢的 ImpromptuInterface NuGet 包的引用。可以使用**管理 NuGet 包**对话框的搜索功能，如下图所示：



3. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Dynamic;
using System.Runtime.CompilerServices;
using System.Threading;
using System.Threading.Tasks;
using ImpromptuInterface;
```

4. 在 Main 方法下面加入以下代码片段：

```
async static Task AsynchronousProcessing()
{
  string result = await GetDynamicAwaitableObject(true);
  Console.WriteLine(result);

  result = await GetDynamicAwaitableObject(false);
  Console.WriteLine(result);
}

static dynamic GetDynamicAwaitableObject(bool
completeSynchronously)
{
  dynamic result = new ExpandoObject();
  dynamic awaiter = new ExpandoObject();

  awaiter.Message = "Completed synchronously";
  awaiter.IsCompleted = completeSynchronously;
  awaiter.GetResult = (Func<string>)(() => awaiter.Message);

  awaiter.OnCompleted = (Action<Action>) ( callback =>
    ThreadPool.QueueUserWorkItem(state => {
      Thread.Sleep(TimeSpan.FromSeconds(1));
      awaiter.Message = GetInfo();
      if (callback != null) callback();
    })
  );

  IAwaiter<string> proxy = Impromptu.ActLike(awaiter);

  result.GetAwaiter = (Func<dynamic>) ( () => proxy );

  return result;
}

static string GetInfo()
{
  return string.Format("Task is running on a thread id {0}.
    Is thread pool thread: {1}",
      Thread.CurrentThread.ManagedThreadId, Thread.CurrentThread.
IsThreadPoolThread);
}

public interface IAwaiter<T> : INotifyCompletion
{
bool IsCompleted { get; }

T GetResult();
}
```

5. 在 Main 方法中加入以下代码片段：

```
Task t = AsynchronousProcessing();
t.Wait();
```

6. 运行程序。

5.10.3　工作原理

这里我们重复了 5.9 节的技巧，但是这次借助于动态表达式，可以使用 NuGet 来实现该目标。NuGet 是一个包含了很多有用的库的包管理器。这次我们将使用一个库来动态地创建封装对象，实现我们需要的接口。

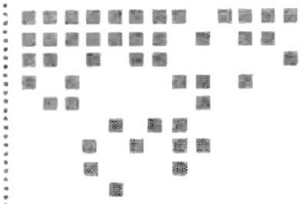
首先我们创建了 ExpandoObject 类型的两个实例，并把它们分配给动态的局部变量。这些变量将成为 awaitable 和 awaiter 对象。由于一个 awaitable 对象只需要拥有 GetAwaiter 方法，提供该方法没有问题。使用 dynamic 关键字组合 ExpandoOjbect 允许我们自定义该对象并通过分配相应的值来添加属性和方法。事实上它是一个字典类型的集合，键类型是 string，值类型是 object。如果你很熟悉 JavaScript 编程语言，你可能会注意到它与 JavaScript 对象很相似。

由于 dynamic 关键字允许我们跳过 C# 的编译时检查。ExpandObject 是以这样的方式编写的：当你给属性分配值时，ExpandObject 创建了一个字典条目，键是属性名，值是赋予的任何值。当尝试获取属性值时，会在字典中查找并提供存储在相应的字典条目中的值。如果该值是 Action 或 Func 类型，我们实际上存储了一个委托，它可以当做方法使用。因此，ExpandoObject 与 dynamic 类型的组合允许我们创建一个对象并动态地赋予其属性和方法。

现在我们需要构造自定义的 awaiter 和 awaitable 对象。先从 awaiter 开始。首先提供一个名为 Message 的属性并赋予初始值，然后使用 Func<string> 类型定义了 GetResult 方法，并分配一个 lambda 表达式，该表达式返回 Message 属性值。接下来实现 IsCompleted 属性。如果其值为 true，则跳过剩下的工作并处理存储在 result 局部变量中的 awaitable 对象。我们只需要添加一个方法用于返回该 dynamic 对象并从该对象返回 awaiter 对象。我们可以使用 result 作为 await 表达式。然而，它将会同步运行。

主要的挑战是在动态对象中实现异步处理。C# 语言规格说明规定 awaiter 必须实现 INotifyCompletion 或 ICriticalNotifyCompletion 接口，但是 ExpandoObject 却没有。甚至当我们动态地实现 OnCompleted 方法并添加到 awaiter 对象时，这仍然行不通，因为该对象没有实现上面提到的任何一个接口。

为了解决该问题，我们使用了 NuGet 提供的 ImpromptuInterface 库。它允许我们使用 Impromptu.ActLike 方法来动态地创建代理对象，该对象将实现任何需要的接口。如果我们尝试创建一个实现了 INotifyCompletion 接口的代理，仍然行不通，因为该代理对象不再是动态的，并且该接口只有 OnCompleted 方法，但没有 IsCompleted 属性或 GetResult 方法。作为最后的解决办法，我们定义了一个泛型接口，IAwaiter<T>，它实现了 INotifyCompletion 并添加了所有需要的属性和方法。现在，我们使用它生成代理并修改 result 对象来从 GetAwaiter 方法返回一个代理，而不是返回 awaiter 对象。现在程序可以工作了，我们构造了一个在运行时完全动态的 awaitable 对象。



Chapter 6 第 6 章

使用并发集合

在本章中，我们将学习 .NET Framework 基础类库中与并发编程相关的不同的数据结构。你将学到以下内容：

- 使用 ConcurrentDictionary
- 使用 ConcurrentQueue 实现异步处理
- 改变 ConcurrentStack 异步处理顺序
- 使用 ConcurrentBag 创建一个可扩展的爬虫
- 使用 BlockingCollection 进行异步处理

6.1 简介

编程需要对基本的数据结构和算法有所了解。程序员为并发情况选择最合适的数据结构，那就需要知道很多事情，例如算法运行时间、空间复杂度，以及大写 O 标记法等。在不同的广为人知的场景中，我们总知道哪种数据结构更高效。

对于并行计算，我们需要使用适当的数据结构。这些数据结构具备可伸缩性，尽可能地避免锁，同时还能提供线程安全的访问。.NET framework 版本 4 引入了 System.Collections.Concurrent 命名空间，其中包含了一些数据结构。在本章中，我们将展示这些数据结构并通过简单的例子来说明如何使用它们。

先从 ConcurrentQueue 开始。该集合使用了**原子的比较和交换**（Compare and Swap，简称 CAS）操作，以及 SpinWait 来保证线程安全。它实现了一个**先进先出**（First In First

Out，简称 FIFO）的集合，这意味着元素出队列的顺序与加入队列的顺序是一致的。可以调用 Enqueue 方法向队列中加入元素。TryDequeue 方法试图取出队列中的第一个元素，而 TryPeek 方法则试图得到第一个元素但并不从队列中删除该元素。

ConcurrentStack 的实现也没有使用任何锁，只采用了 CAS 操作。它是一个**后进先出**（Last In First Out，简称 LIFO）的集合，这意味着最近添加的元素会先返回。可以使用 Push 和 PushRange 方法添加元素，使用 TryPop 和 TryPopRange 方法获取元素，以及使用 TryPeek 方法检查元素。

ConcurrentBag 是一个支持重复元素的无序集合。它针对这样以下情况进行了优化，即多个线程以这样的方式工作：每个线程产生和消费自己的任务，极少与其他线程的任务交互（如果要交互则使用锁）。添加元素使用 Add 方法，检查元素使用 TryPeek 方法，获取元素使用 TryTake 方法。



请避免使用上面提及的集合的 Count 属性。实现这些集合使用的是链表，Count 操作的时间复杂度为 O（N）。如果想检查集合是否为空，请使用 IsEmpty 属性，其时间复杂度为 O（1）。

ConcurrentDictionary 是一个线程安全的字典集合的实现。对于读操作无需使用锁。但是对于写操作则需要锁。该并发字典使用多个锁，在字典桶之上实现了一个细粒度的锁模型。使用参数 concurrencyLevel 可以在构造函数中定义锁的数量，这意味着预估的线程数量将并发地更新该字典。



由于并发字典使用锁，所以一些操作需要获取该字典中的所有锁。如果没必要请避免使用以下操作：Count、IsEmpty、Keys、Values、CopyTo 及 ToArray。

BlockingCollection 是对 IProducerConsumerCollection 泛型接口的实现的一个高级封装。它有很多先进的功能来实现管道场景，即当你有一些步骤需要使用之前步骤运行的结果时。BlockingCollectione 类支持如下功能：分块、调整内部集合容量、取消集合操作、从多个块集合中获取元素。

并行算法有可能非常复杂，并且或多或少涵盖了这些并行集合，甚至都可以写一本单独的书了。这里我只说明使用并发集合的最简单的例子。

6.2　使用 ConcurrentDictionary

本节展示了一个非常简单的场景，比较在单线程环境中使用通常的字典集合与使用并发字典的性能。

6.2.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter6\Recipe1 目录中。

6.2.2 实现方式

请执行以下步骤来了解通常的字典集合与并发字典的性能差异：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Diagnostics;
```

3. 在 Main 方法下面加入以下代码片段：

```
const string Item = "Dictionary item";
public static string CurrentItem;
```

4. 在 Main 方法中加入以下代码片段：

```
var concurrentDictionary = new ConcurrentDictionary<int,
  string>();
var dictionary = new Dictionary<int, string>();

var sw = new Stopwatch();

sw.Start();
for (int i = 0; i < 1000000; i++)
{
  lock (dictionary)
  {
    dictionary[i] = Item;
  }
}

sw.Stop();
Console.WriteLine("Writing to dictionary with a lock: {0}",
  sw.Elapsed);

sw.Restart();
for (int i = 0; i < 1000000; i++)
{
  concurrentDictionary[i] = Item;
}
sw.Stop();
Console.WriteLine("Writing to a concurrent dictionary:
  {0}", sw.Elapsed);

sw.Restart();
```

```
for (int i = 0; i < 1000000; i++)
{
  lock (dictionary)
  {
    CurrentItem = dictionary[i];
  }
}
sw.Stop();
Console.WriteLine("Reading from dictionary with a lock:
  {0}", sw.Elapsed);

sw.Restart();
for (int i = 0; i < 1000000; i++)
{
  CurrentItem = concurrentDictionary[i];
}
sw.Stop();
Console.WriteLine("Reading from a concurrent dictionary:
  {0}", sw.Elapsed);
```

5. 运行程序。

6.2.3　工作原理

当程序启动时我们创建了两个集合，其中一个是标准的字典集合，另一个是新的并发字典集合。然后采用锁的机制向标准的字典中添加元素，并测量完成 100 万次迭代的时间。同样也采用同样的场景来测量 ConcurrentDictionary 的性能，最后比较从两个集合中获取值的性能。

通过这个非常简单的场景，我们发现 ConcurrentDictionary 写操作比使用锁的通常的字典要慢得多，而读操作则要快些。因此如果对字典需要大量的线程安全的读操作，ConcurrentDictionary 是最好的选择。



如果你对字典只需要多线程访问只读元素，则没必要执行线程安全的读操作。在此场景中最好只使用通常的字典或 ReadOnlyDictionary 集合。

ConcurrentDictionary 的实现使用了**细粒度锁**（fine-grained locking）技术，这在多线程写入方面比使用锁的通常的字典（也被称为**粗粒度锁**）的可伸缩性更好。正如本例中所示，当只用一个线程时，并发字典非常慢，但是扩展到 5 到 6 个线程（如果有足够的 CPU 核心来同时运行它们），并发字典的性能会更好。

6.3　使用 ConcurrentQueue 实现异步处理

本节将展示创建能被多个工作者异步处理的一组任务的例子。

6.3.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter6\Recipe2 目录中。

6.3.2 实现方式

请执行以下步骤来创建能被多个工作者异步处理的一套任务：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Collections.Concurrent;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
static async Task RunProgram()
{
  var taskQueue = new ConcurrentQueue<CustomTask>();
  var cts = new CancellationTokenSource();

  var taskSource = Task.Run(() => TaskProducer(taskQueue));

  Task[] processors = new Task[4];
  for (int i = 1; i <= 4; i++)
  {
    string processorId = i.ToString();
    processors[i-1] = Task.Run(
    () => TaskProcessor(taskQueue, "Processor " +
      processorId, cts.Token));
  }

  await taskSource;
  cts.CancelAfter(TimeSpan.FromSeconds(2));

  await Task.WhenAll(processors);
}

static async Task TaskProducer
  (ConcurrentQueue<CustomTask> queue)
{
  for (int i = 1; i <= 20; i++)
  {
    await Task.Delay(50);
    var workItem = new CustomTask {Id = i};
    queue.Enqueue(workItem);
    Console.WriteLine("Task {0} has been posted",
      workItem.Id);
  }
}
```

```
static async Task TaskProcessor(
  ConcurrentQueue<CustomTask> queue, string name,
  CancellationToken token)
{
  CustomTask workItem;
  bool dequeueSuccesful = false;

  await GetRandomDelay();
  do
  {
    dequeueSuccesful = queue.TryDequeue(out workItem);
    if (dequeueSuccesful)
    {
    Console.WriteLine("Task {0} has been processed by {1}",
      workItem.Id, name);
    }

    await GetRandomDelay();
  }
  while (!token.IsCancellationRequested);
}

static Task GetRandomDelay()
{
  int delay = new Random(DateTime.Now.Millisecond).Next(1,
    500);
  return Task.Delay(delay);
}

class CustomTask
{
  public int Id { get; set; }
}
```

4. 在 Main 方法中加入以下代码片段：

```
Task t = RunProgram();
t.Wait();
```

5. 运行程序。

6.3.3　工作原理

当程序运行时，我们使用 ConcurrentQueue 集合实例创建了一个任务队列。然后创建了一个取消标志，它是用来在我们将任务放入队列后停止工作的。接下来启动了一个单独的工作线程来将任务放入任务队列中。该部分为异步处理产生了工作量。

现在定义该程序中消费任务的部分。我们创建了四个工作者，它们会随机等待一段时间，然后从任务队列中获取一个任务，处理该任务，一直重复整个过程直到我们发出取消标志信号。最后，我们启动产生任务的线程，等待该线程完成。然后使用取消标志给消费者发信号我们完成了工作。最后一步将等待所有的消费者完成。

我们看到队列中的任务按从前到后的顺序被处理，但一个后面的任务是有可能会比前面的任务先处理的，因为我们有四个工作者独立地运行，而且任务处理时间并不是恒定的。我们看到访问该队列是线程安全的，没有一个元素会被提取两次。

6.4 改变 ConcurrentStack 异步处理顺序

本节是前一小节的细微修改版。我们又一次创建了被多个工作者异步处理的一组任务，但是这次使用 ConcurrentStack 来实现并看看有什么不同。

6.4.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter6\Recipe3 目录中。

6.4.2 实现方式

请执行以下步骤来了解使用 ConcurrentStack 实现一组任务的处理过程：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Collections.Concurrent;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
static async Task RunProgram()
{
  var taskStack = new ConcurrentStack<CustomTask>();
  var cts = new CancellationTokenSource();

  var taskSource = Task.Run(() => TaskProducer(taskStack));

  Task[] processors = new Task[4];
  for (int i = 1; i <= 4; i++)
  {
    string processorId = i.ToString();
    processors[i - 1] = Task.Run(
    () => TaskProcessor(taskStack, "Processor " +
      processorId, cts.Token));
  }

  await taskSource;
  cts.CancelAfter(TimeSpan.FromSeconds(2));

  await Task.WhenAll(processors);
```

```
}

static async Task TaskProducer(ConcurrentStack<CustomTask>
  stack)
{
  for (int i = 1; i <= 20; i++)
  {
    await Task.Delay(50);
    var workItem = new CustomTask { Id = i };
    stack.Push(workItem);
    Console.WriteLine("Task {0} has been posted",
      workItem.Id);
  }
}

static async Task TaskProcessor(
  ConcurrentStack<CustomTask> stack, string name,
    CancellationToken token)
{
  await GetRandomDelay();
  do
  {
    CustomTask workItem;
    bool popSuccesful = stack.TryPop(out workItem);
    if (popSuccesful)
    {
    Console.WriteLine("Task {0} has been processed by {1}",
      workItem.Id, name);
    }

    await GetRandomDelay();
  }
  while (!token.IsCancellationRequested);
}

static Task GetRandomDelay()
{
  int delay = new Random(DateTime.Now.Millisecond).Next(1,
    500);
  return Task.Delay(delay);
}

class CustomTask
{
  public int Id { get; set; }
}
```

4. 在 Main 方法中加入以下代码片段：

```
Task t = RunProgram();
t.Wait();
```

5. 运行程序。

6.4.3 工作原理

当程序运行时，我们创建了一个 ConcurrentStack 集合的实例。其余的代码与前一小节

中几乎一样，唯一不同之处是我们对并发堆栈使用 Push 和 TryPop 方法，而对并发队列使用 Enqueue 和 TryDequeue 方法。

现在可以看到任务处理的顺序被改变了。堆栈是一个 LIFO 集合，工作者先处理最近的任务。在并发队列中，任务被处理的顺序与被添加的顺序几乎一致。这意味着根据工作者的数量，我们必将在一定时间窗内处理先被创建的任务。而在堆栈中，早先创建的任务具有较低的优先级，而且直到生产者停止向堆栈中放入更多任务后，该任务才有可能被处理。这种行为是确定的，最好在该场景下使用队列。

6.5 使用 ConcurrentBag 创建一个可扩展的爬虫

本节展示了在多个独立的既可生产工作又可消费工作的工作者间如何扩展工作量。

6.5.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter6\Recipe4 目录中。

6.5.2 实现方式

以下步骤演示了在一堆即可产生工作又可处理工作的相互独立的工作者间如何扩展工作量：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
static Dictionary<string, string[]> _contentEmulation = new
  Dictionary<string, string[]>();

static async Task RunProgram()
{
  var bag = new ConcurrentBag<CrawlingTask>();

  string[] urls = new[] {"http://microsoft.com/",
    "http://google.com/", "http://facebook.com/",
    "http://twitter.com/"};

  var crawlers = new Task[4];
  for (int i = 1; i <= 4; i++)
  {
```

```
    string crawlerName = "Crawler " + i.ToString();
    bag.Add(new CrawlingTask { UrlToCrawl = urls[i-1],
      ProducerName = "root"});
    crawlers[i - 1] = Task.Run(() => Crawl(bag,
      crawlerName));
  }

  await Task.WhenAll(crawlers);
}

static async Task Crawl(ConcurrentBag<CrawlingTask> bag,
  string crawlerName)
{
  CrawlingTask task;
  while (bag.TryTake(out task))
  {
    IEnumerable<string> urls = await
      GetLinksFromContent(task);
    if (urls != null)
    {
      foreach (var url in urls)
      {
        var t = new CrawlingTask
        {
          UrlToCrawl = url,
          ProducerName = crawlerName
        };

      bag.Add(t);
      }
    }

  Console.WriteLine("Indexing url {0} posted by {1} is
    completed by {2}!",
      task.UrlToCrawl, task.ProducerName, crawlerName);
  }
}

static async Task<IEnumerable<string>>
  GetLinksFromContent(CrawlingTask task)
{
  await GetRandomDelay();

  if (_contentEmulation.ContainsKey(task.UrlToCrawl))
    return _contentEmulation[task.UrlToCrawl];

  return null;
}

static void CreateLinks()
{
  _contentEmulation["http://microsoft.com/"] = new [] {
    "http://microsoft.com/a.html",
    "http://microsoft.com/b.html" };
  _contentEmulation["http://microsoft.com/a.html"] = new[]
    { "http://microsoft.com/c.html",
    "http://microsoft.com/d.html" };
```

```
    _contentEmulation["http://microsoft.com/b.html"] = new[]
      { "http://microsoft.com/e.html" };

    _contentEmulation["http://google.com/"] = new[] {
      "http://google.com/a.html", "http://google.com/b.html"
      };
    _contentEmulation["http://google.com/a.html"] = new[] {
      "http://google.com/c.html", "http://google.com/d.html"
      };
    _contentEmulation["http://google.com/b.html"] = new[] {
      "http://google.com/e.html", "http://google.com/f.html"
      };
    _contentEmulation["http://google.com/c.html"] = new[] {
      "http://google.com/h.html", "http://google.com/i.html"
      };

    _contentEmulation["http://facebook.com/"] = new [] {
      "http://facebook.com/a.html",
      "http://facebook.com/b.html" };
    _contentEmulation["http://facebook.com/a.html"] = new[] {
      "http://facebook.com/c.html",
      "http://facebook.com/d.html" };
    _contentEmulation["http://facebook.com/b.html"] = new[] {
      "http://facebook.com/e.html" };
    _contentEmulation["http://twitter.com/"] = new[] {
      "http://twitter.com/a.html",
      "http://twitter.com/b.html" };
    _contentEmulation["http://twitter.com/a.html"] = new[] {
      "http://twitter.com/c.html",
      "http://twitter.com/d.html" };
    _contentEmulation["http://twitter.com/b.html"] = new[] {
      "http://twitter.com/e.html" };
    _contentEmulation["http://twitter.com/c.html"] = new[] {
      "http://twitter.com/f.html",
      "http://twitter.com/g.html" };
    _contentEmulation["http://twitter.com/d.html"] = new[] {
      "http://twitter.com/h.html" };
    _contentEmulation["http://twitter.com/e.html"] = new[] {
      "http://twitter.com/i.html" };
}

static Task GetRandomDelay()
{
  int delay = new Random(DateTime.Now.Millisecond)
    .Next(150, 200);
  return Task.Delay(delay);
}

class CrawlingTask
{
  public string UrlToCrawl { get; set; }

  public string ProducerName { get; set; }
}
```

4. 在 Main 方法中加入以下代码片段：

```
CreateLinks();
Task t = RunProgram();
t.Wait();
```

5. 运行程序。

6.5.3　工作原理

该程序模拟了使用多个网络爬虫进行网页索引的场景。网络爬虫是这样一个程序：它使用网页地址打开一个网页，索引该网页内容，尝试访问该页面包含的所有链接，并且也索引这些链接页面。刚开始，我们定义了一个包含不同网页 URL 的字典。该字典模拟了包含其他页面链接的网页。该实现非常简单，并不关心索引已经访问过的页面，但正因为它如此简单我们才可以关注并行工作负载。

接着创建了一个并发包，其中包含爬虫任务。我们创建了四个爬虫，并且给每个爬虫都提供了一个不同的网站根 URL。然后等待所有爬虫完成工作。现在每个爬虫开始检索提供给它的网站 URL。我们通过等待一个随机事件来模拟网络 I/O 处理。如果页面包含的 URL 越多，爬虫向包中放入的任务也会越多。然后检查包中是否还有任何需要爬虫处理的任务，如果没有说明爬虫完成了工作。

如果检查前四个根 URL 后的第一行输出内容，我们将看到被爬虫 N 放置的任务通常会被同一个爬虫处理。然而，接下来的行则会不同。这是因为 ConcurrentBag 内部针对多个线程既可以添加元素又可以删除元素的场景进行了优化。实现方式是每个线程使用自己的本地队列的元素，所以使用该队列时无需任何锁。只有当本地队列中没有任何元素时，我们才执行一些锁定操作并尝试从其他线程的本地队列中“偷取”工作。这种行为有助于在所有工作者间分发工作并避免使用锁。

6.6　使用 BlockingCollection 进行异步处理

本节将描述如何使用 BlockingCollection 来简化实现异步处理的工作负载。

6.6.1　准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter6\Recipe5 目录中。

6.6.2　实现方式

请执行以下步骤来了解 BlockingCollection 如何简化实现异步处理的工作负载。

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Collections.Concurrent;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
static async Task RunProgram
  (IProducerConsumerCollection<CustomTask> collection =
  null)
{
  var taskCollection = new
    BlockingCollection<CustomTask>();
  if (null != collection)
  taskCollection= new
    BlockingCollection<CustomTask>(collection);

  var taskSource = Task.Run(() =>
    TaskProducer(taskCollection));

  Task[] processors = new Task[4];
  for (int i = 1; i <= 4; i++)
  {
    string processorId = "Processor " + i;
    processors[i - 1] = Task.Run(
      () => TaskProcessor(taskCollection, processorId));
  }

  await taskSource;

  await Task.WhenAll(processors);
}

static async Task TaskProducer
  (BlockingCollection<CustomTask> collection)
{
  for (int i = 1; i <= 20; i++)
  {
    await Task.Delay(20);
    var workItem = new CustomTask { Id = i };
    collection.Add(workItem);
    Console.WriteLine("Task {0} have been posted",
    workItem.Id);
  }
  collection.CompleteAdding();
}

static async Task TaskProcessor(
  BlockingCollection<CustomTask> collection, string name)
{
  await GetRandomDelay();
  foreach (CustomTask item in
    collection.GetConsumingEnumerable())
  {
    Console.WriteLine("Task {0} have been processed by
    {1}", item.Id, name);
    await GetRandomDelay();
  }
```

```
}

static Task GetRandomDelay()
{
  int delay = new Random(DateTime.Now.Millisecond).Next(1,
    500);
  return Task.Delay(delay);
}

class CustomTask
{
  public int Id { get; set; }
}
```

4. 在 Main 方法中加入以下代码片段：

```
Console.WriteLine("Using a Queue inside of
  BlockingCollection");
Console.WriteLine();
Task t = RunProgram();
t.Wait();

Console.WriteLine();
Console.WriteLine("Using a Stack inside of
  BlockingCollection");
Console.WriteLine();
t = RunProgram(new ConcurrentStack<CustomTask>());
t.Wait();
```

5. 运行程序。

6.6.3　工作原理

先说第一个场景，这里我们使用了 BlockingCollection 类，它带来了很多优势。首先，我们能够改变任务存储在阻塞集合中的方式。默认情况下它使用的是 ConcurrentQueue 容器，但是我们能够使用任何实现了 IProducerConsumerCollection 泛型接口的集合。为了演示该点，我们运行了该程序两次，第二次时使用 ConcurrentStack 作为底层集合。

工作者通过对阻塞集合迭代调用 GetConsumingEnumerable 方法来获取工作项。如果在该集合中没有任何元素，迭代器会阻塞工作线程直到有元素被放置到集合中。当生产者调用集合的 CompleteAdding 时该迭代周期会结束。这标志着工作完成了。



这里很容易犯一个错误，即对 BlockingCollection 进行迭代，因为它自身实现了 IEnumerable 接口。不要忘记使用 GetConsumingEnumerable，否则你迭代的只是集合的“快照”，这并不是期望的程序行为。

工作量生产者将任务插入到 BlockingCollection 然后调用 CompleteAdding 方法，这会使所有工作者完成工作。现在在程序输出中我们看到两个结果序列，演示了并发队列和堆栈集合的不同之处。



Chapter 7 第 7 章

使用 PLINQ

在本章中我们将回顾不同的并行编程范例，例如任务和数据并行度，以及数据平行度和并行 LINQ 查询的基础知识。你将学到以下内容：

- 使用 Parallel 类
- 并行化 LINQ 查询
- 调整 PLINQ 查询的参数
- 处理 PLINQ 查询中的异常
- 管理 PLINQ 查询中的数据分区
- 为 PLINQ 查询创建一个自定义的聚合器

7.1 简介

.NET Framework 库中有个子集叫做并行库，通常被称为**并行框架扩展**（Parallel Framework Extensions，简称 **PFX**），这是这些库非常早期的版本的名称。并行库随着 .NET Framework 4.0 一起发布，包含三大主要部分：

- **任务并行库**（TPL）
- 并发集合
- 并行 LINQ（或 PLINQ）

通过本书，我们学习了如何并行地运行多个任务以及如何同步这些任务。事实上我们将程序分割成一组任务并使用不同的线程来运行不同的任务。这种方式被称为**任务并行**（task

parallelism)，目前我们只学习了任务并行。

想象一下我们有一个程序针对一组大数据进行重量级运算。并行运行该程最容易的方式是将该组数据分割成较小的数据块，对这些数据块进行并行计算，然后聚合这些计算结果。这种编程模型称为**数据并行**（data parallelism）。

任务并行是最底层的抽象层。我们将程序定义为任务的组合，显式地定义这些任务如何组合。由此方式组成的程序会非常复杂和细节化。并行操作被定义在该程序的不同位置，随着并行操作的增长，程序变得越来越难理解和维护。采用这种方式来并行程序被称为**无结构的并行**（unstructured parallelism）。这就是我们为复杂的并行逻辑付出的代价。

然而，当我们有较简单的程序逻辑时，我们可以将更多的并行细节推给 PFX 库和 C# 编译器。例如，我们可以说，“我想以并行方式运行这三个方法，但我不关心是如何实现并行的，让 .NET 基础设施决定细节。”这产生了一个抽象层使得我们不用提供一个关于如何实现并行的细节描述。这种方式被称为**结构并行**（structured parallelism），因为并行通常是一组声明，并且在程序中每个并行情况并定义在确切的地方。



这可能导致一种印象，即无结构并行是一种不好的实践，应该始终使用结构并行替代它。我想强调这一点是不对的。结构并行确实更易维护，应该尽可能地使用，但是它并不是万能的。通常有很多情况我们不能简单地使用结构并行，那么以非结构化的方式使用 TPL 任务并行也是完全可以的。

任务并行库中有一个名为 Parallel 的类，其提供了一组 API 用来实现结构并行。它仍然是 TPL 的一部分，我们在本章介绍它的原因是它是从较低的抽象层向较高的抽象层过渡的完美例子。当使用 Parallel 类的 API 时，我们无需提供分割工作的细节。但是我们仍要显式定义如何从分割的结果中得到单个结果。

PLINQ 具有最高级抽象。它自动将数据分割为数据块，并且决定是否真的需要并行化查询，或者使用通常的顺序查询处理更高效。PLINQ 基础设施会将分割任务的执行结果组合到一起。有很多选项可供程序员来优化查询，使用尽可能高的性能获取结果。

在本章中我们将涵盖 Parallel 类的用法以及很多不同的 PLINQ 选项，例如让 LINQ 查询并行化，设置异常模型及设置 PLINQ 查询的并行等级，处理查询项的顺序，以及处理 PLINQ 异常。我们也会学习如何管理 PLINQ 查询的数据分割。

7.2　使用 Parallel 类

本节展示了如何使用 Parallel 类的 API。我们将学习如何并行地调用方法，如何执行并行的循环，以及调整并行机制。

7.2.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter7\Recipe1 目录中。

7.2.2 实现方式

请执行以下步骤来使用 Parallel 类并行地调用方法，执行并行循环，调整并行机制：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
static string EmulateProcessing(string taskName)
{
  Thread.Sleep(TimeSpan.FromMilliseconds(
    new Random(DateTime.Now.Millisecond).Next(250, 350)));
  Console.WriteLine("{0} task was processed on a thread id
    {1}",taskName, Thread.CurrentThread.ManagedThreadId);
  return taskName;
}
```

4. 在 Main 方法中加入以下代码片段：

```
Parallel.Invoke(
  () => EmulateProcessing("Task1"),
  () => EmulateProcessing("Task2"),
  () => EmulateProcessing("Task3")
);

var cts = new CancellationTokenSource();
var result = Parallel.ForEach(
  Enumerable.Range(1, 30),
  new ParallelOptions
  {
    CancellationToken = cts.Token,
    MaxDegreeOfParallelism = Environment.ProcessorCount,
    TaskScheduler = TaskScheduler.Default
  },
  (i, state) =>
  {
    Console.WriteLine(i);
    if (i == 20)
    {
      state.Break();
      Console.WriteLine("Loop is stopped: {0}",
```

```
            state.IsStopped);
        }
    });

Console.WriteLine("---");
Console.WriteLine("IsCompleted: {0}", result.IsCompleted);
Console.WriteLine("Lowest break iteration: {0}",
    result.LowestBreakIteration);
```

5. 运行程序。

7.2.3　工作原理

该程序演示了 Parallel 类的不同功能。与在任务并行库中定义任务的方式相比，调用 Invoke 方法可以免去很多麻烦就可实现并行地运行多个任务。Invoke 方法会阻塞其他线程直到所有的任务都被完成，这是一个非常常见的方面使用 Invoke 方法的场景。

下一个功能是并行循环，使用 For 和 ForEach 方法来定义循环。由 ForEach 方法与 For 方法非常相似，我们将仔细讲解 ForEach 方法。并行 ForEach 循环可以通过给每个集合项应用一个 action 委托的方式，实现并行地处理任何 IEnumerable 集合。我们可以提供几种选项，自定义并行行为，并得到一个结果来说明循环是否成功完成。

可以给 ForEach 方法提供一个 ParallelOptions 类的实例来控制并行循环。其允许我们使用 CancellationToken 取消循环，限制最大并行度（并行运行的最大操作数），还可以提供一个自定义的 TaskScheduler 类来调度任务。Action 可以接受一个附加的 ParallelLoopState 参数，可用于从循环中跳出或者检查当前循环的状态。

使用 ParallelLoopState 有两种方式停止并行循环。既可以使用 Break 方法，也可以使用 Stop 方法。Stop 方法告诉循环停止处理任何工作，并设置并行循环状态属性 IsStopped 值为 true。Break 方法停止其之后的迭代，但之前的迭代还要继续工作。在那种情况下，循环结果的 LowestBreakIteration 属性将会包含当 Break 方法被调用时的最低循环次数。

7.3　并行化 LINQ 查询

本节将描述如何使用 PLINQ 来并行化查询，以及如何将并行查询改为顺序处理。

7.3.1　准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter7\Recipe2 目录中。

7.3.2 实现方式

请执行以下步骤来使用 PLINQ 并行化查询，以及将并行查询改为顺序处理：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void PrintInfo(string typeName)
{
  Thread.Sleep(TimeSpan.FromMilliseconds(150));
  Console.WriteLine("{0} type was printed on a thread id
    {1}", typeName, Thread.CurrentThread.ManagedThreadId);
}

static string EmulateProcessing(string typeName)
{
  Thread.Sleep(TimeSpan.FromMilliseconds(150));
  Console.WriteLine("{0} type was processed on a thread id
    {1}",typeName, Thread.CurrentThread.ManagedThreadId);
  return typeName;
}

static IEnumerable<string> GetTypes()
{
  return from assembly in
    AppDomain.CurrentDomain.GetAssemblies()
    from type in assembly.GetExportedTypes()
    where type.Name.StartsWith("Web")
    select type.Name;
}
```

4. 在 Main 方法中加入以下代码片段：

```
var sw = new Stopwatch();
sw.Start();
var query = from t in GetTypes()
  select EmulateProcessing(t);

foreach (string typeName in query)
{
  PrintInfo(typeName);
}
sw.Stop();
Console.WriteLine("---");
Console.WriteLine("Sequential LINQ query.");
```

```
Console.WriteLine("Time elapsed: {0}", sw.Elapsed);
Console.WriteLine("Press ENTER to continue....");
Console.ReadLine();
Console.Clear();
sw.Reset();

sw.Start();
var parallelQuery = from t in
  ParallelEnumerable.AsParallel(GetTypes())
  select EmulateProcessing(t);

foreach (string typeName in parallelQuery)
{
  PrintInfo(typeName);
}
sw.Stop();
Console.WriteLine("---");
Console.WriteLine("Parallel LINQ query. The results are
  being merged on a single thread");
Console.WriteLine("Time elapsed: {0}", sw.Elapsed);
Console.WriteLine("Press ENTER to continue....");
Console.ReadLine();
Console.Clear();
sw.Reset();

sw.Start();
parallelQuery = from t in GetTypes().AsParallel()
  select EmulateProcessing(t);

parallelQuery.ForAll(PrintInfo);

sw.Stop();
Console.WriteLine("---");
Console.WriteLine("Parallel LINQ query. The results are
  being processed in parallel");
Console.WriteLine("Time elapsed: {0}", sw.Elapsed);
Console.WriteLine("Press ENTER to continue....");
Console.ReadLine();
Console.Clear();
sw.Reset();

sw.Start();
query = from t in GetTypes().AsParallel().AsSequential()
  select EmulateProcessing(t);

foreach (var typeName in query)
{
  PrintInfo(typeName);
}

sw.Stop();
```

```
Console.WriteLine("---");
Console.WriteLine("Parallel LINQ query, transformed into
  sequential.");
Console.WriteLine("Time elapsed: {0}", sw.Elapsed);
Console.WriteLine("Press ENTER to continue....");
Console.ReadLine();
Console.Clear();
```

5. 运行程序。

7.3.3 工作原理

当程序运行时，我们创建了一个 LINQ 查询，其使用反射 API 来查询加载到当前应用程序域中的所有组件中名称以“ Web”开头的类型。我们使用 EmulateProcessing 方法模拟处理每个项时间的延迟，并使用 PrintInfo 方法打印结果。我们也使用了 Stopwatch 类来测量每个查询的执行时间。

首先我们运行了一个通常的顺序 LINQ 查询。此时并没有并行化，所有任何操作都运行在当前线程。该查询的第二版显式地使用了 ParallelEnumerable 类。ParallelEnumerable 包含了 PLINQ 的逻辑实现，并且作为 IEnumerable 集合功能的一组扩展方法。通常无需显式地使用该类，在这里是为了演示 PLINQ 的实际工作方式。第二个版本以并行的方式运行 EmulateProcessing 操作。然而，默认情况下结果会被合并到单个线程中，所以查询的执行时间应该比第一个版本少几秒。

第三个版本展示了如何使用 AsParallel 方法来将 LINQ 查询按声明的方式并行化运行。这里我们并不关心实现细节，只是为了说明我们想以并行的方式运行。然而，该版本的关键不同处是我们使用了 ForAll 方法来打印查询结果。打印结果操作与任务被处理的线程是同一个线程，跳过了结果合并步骤。它允许我们也能以并行的方式运行 PrintInfo 方法，甚至该版本运行速度比之前的版本更快。

最后一个例子展示了如何使用 AsSequential 方法将 PLINQ 查询以顺序方式运行。可以看到该查询运行方式与第一个示例完全一样。

7.4 调整 PLINQ 查询的参数

本节展示了使用 PLINQ 查询时如何管理并行处理选项，以及查询执行时这些选项的效果。

7.4.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter7\Recipe3 目录中。

7.4.2　实现方式

请执行以下步骤来了解使用 PLINQ 查询时如何管理并行处理选项，以及这些选项的效果：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static string EmulateProcessing(string typeName)
{
  Thread.Sleep(TimeSpan.FromMilliseconds(
    new Random(DateTime.Now.Millisecond).Next(250,350)));
  Console.WriteLine("{0} type was processed on a thread id
    {1}",typeName, Thread.CurrentThread.ManagedThreadId);
  return typeName;
}

static IEnumerable<string> GetTypes()
{
  return from assembly in
    AppDomain.CurrentDomain.GetAssemblies()
    from type in assembly.GetExportedTypes()
    where type.Name.StartsWith("Web")
    orderby type.Name.Length
    select type.Name;
}
```

4. 在 Main 方法中加入以下代码片段：

```
var parallelQuery = from t in GetTypes().AsParallel()
  select EmulateProcessing(t);

var cts = new CancellationTokenSource();
cts.CancelAfter(TimeSpan.FromSeconds(3));

try
{
  parallelQuery.WithDegreeOfParallelism
    (Environment.ProcessorCount).WithExecutionMode
    (ParallelExecutionMode.ForceParallelism)
    .WithMergeOptions (ParallelMergeOptions.Default)
    .WithCancellation(cts.Token).ForAll(Console.WriteLine);
}
catch (OperationCanceledException)
{
  Console.WriteLine("---");
```

```
    Console.WriteLine("Operation has been canceled!");
}

Console.WriteLine("---");
Console.WriteLine("Unordered PLINQ query execution");
var unorderedQuery = from i in ParallelEnumerable.Range(1,
  30) select i;

foreach (var i in unorderedQuery)
{
  Console.WriteLine(i);
}

Console.WriteLine("---");
Console.WriteLine("Ordered PLINQ query execution");
var orderedQuery = from i in ParallelEnumerable.Range(1,
  30).AsOrdered() select i;

foreach (var i in orderedQuery)
{
  Console.WriteLine(i);
}
```

5. 运行程序。

7.4.3 工作原理

该程序演示了多个可供程序员使用的非常有用的 PLINQ 选项。我们先创建了一个 PLINQ 查询，然后又创建了一个调整了 PLINQ 选项的查询。

先从取消选项开始。接受一个取消标志对象的 WithCancellation 方法可用于取消查询。本例中我们三秒后发出取消标志信号，这导致查询中抛出 OperationCanceledException 异常，并且取消了剩余的工作。

然后可以为查询指定并行度。这是被用于执行查询时实际并行分割数。在第一节中我们使用了 Parallel.ForEach 循环，其拥有最大并行度选项。该选项与本节中的不一样，这是因为它指定了一个最大的分割值，但如果基础设施决定最好使用较少的并行度以节省资源和提高性能，那么并行度会小于最大值。

另一个有意思的选项是使用 WithExecutionMode 方法来重载查询执行的模式。PLINQ 基础设施如果认为并行化某查询只会增加工作量并且运行更慢，那么将会以顺序模式执行该查询。但我们可以强制该查询以并行的方式运行。

可以使用 WithMergeOptions 方法调整对查询结果的处理。默认模式是 PLINQ 基础设施在查询返回结果之前会缓存一定数量的结果。如果查询花费了大量的时间，更合理的方式是关闭结果缓存从而尽可能快地得到结果。

最后一个选项是 AsOrdered 方法。当使用并行执行时，集合中的项有可能不是被顺序处理的。集合中稍后的项可能会比稍前的项先处理。为了防止该情况，我们可以显式的对并行查询调用 AsOrdered 方法，来告诉 PLINQ 基础设施我们打算按项在集合中的顺序来进行处理。

7.5　处理 PLINQ 查询中的异常

本节将描述如何处理 PLINQ 查询中的异常。

7.5.1　准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter7\Recipe4 目录中。

7.5.2　实现方式

请执行以下步骤来了解如何处理 PLINQ 查询中的异常：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Collections.Generic;
using System.Linq;
```

3. 在 Main 方法中加入以下代码片段：

```
IEnumerable<int> numbers = Enumerable.Range(-5, 10);

var query = from number in numbers
  select 100 / number;

try
{
  foreach(var n in query)
    Console.WriteLine(n);
}
catch (DivideByZeroException)
{
  Console.WriteLine("Divided by zero!");
}

Console.WriteLine("---");
Console.WriteLine("Sequential LINQ query processing");
Console.WriteLine();

var parallelQuery = from number in numbers.AsParallel()
```

```
    select 100 / number;

try
{
  parallelQuery.ForAll(Console.WriteLine);
}
catch (DivideByZeroException)
{
  Console.WriteLine("Divided by zero - usual exception
    handler!");
}
catch (AggregateException e)
{
  e.Flatten().Handle(ex =>
  {
    if (ex is DivideByZeroException)
      {
      Console.WriteLine("Divided by zero - aggregate
        exception handler!");
      return true;
      }

    return false;
  });
}

Console.WriteLine("---");
Console.WriteLine("Parallel LINQ query processing and
  results merging");
```

4. 运行程序。

7.5.3 工作原理

首先我们对从 -5 到 4 的一组数字运行了一个平常的 LINQ 查询。当除以 0 时，得到了 DivideByZeroException 异常并使用 try/catch 代码块这种通常的方式来处理该异常。

然而，当使用 AsParallel 运行，我们得到的却是 AggregateException，因为现在是以并行的方式运行，受到了幕后的任务基础设施的影响。AggregateException 将包含运行 PLINQ 查询期间发生的所有异常。我们可以使用 Flatten 和 Handle 方法来处理内部的 DivideByZeroException 类，在 5.6 节做过讲解。



处理聚合异常时，很容易忘记在其中会有多于一个的内部异常这一很常见的情况。如果你忘记处理里面的所有异常，该异常会向上传递导致应用程序停止工作。

7.6 管理 PLINQ 查询中的数据分区

本节展示了如何创建一个非常基础的自定义分区策略，用来以指定的方式并行化 LINQ 查询。

7.6.1　准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter7\Recipe5 目录中。

7.6.2　实现方式

请执行以下操作来学习如何创建一个非常基本的自定义分区策略来并行化 LINQ 查询：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static void PrintInfo(string typeName)
{
  Thread.Sleep(TimeSpan.FromMilliseconds(150));
  Console.WriteLine("{0} type was printed on a thread id
    {1}",typeName, Thread.CurrentThread.ManagedThreadId);
}

static string EmulateProcessing(string typeName)
{
  Thread.Sleep(TimeSpan.FromMilliseconds(150));
  Console.WriteLine("{0} type was processed on a thread id
    {1}. Has {2} length.",typeName,
    Thread.CurrentThread.ManagedThreadId, typeName.Length %
    2 == 0 ? "even" : "odd");
  return typeName;
}

static IEnumerable<string> GetTypes()
{
  var types = AppDomain.CurrentDomain
    .GetAssemblies()
    .SelectMany(a => a.GetExportedTypes());

  return from type in types where
    type.Name.StartsWith("Web")
    select type.Name;
}

public class StringPartitioner : Partitioner<string>
{
```

```
  private readonly IEnumerable<string> _data;

  public StringPartitioner(IEnumerable<string> data)
  {
    _data = data;
  }

  public override bool SupportsDynamicPartitions
  {
    get
    {
     return false;
    }
  }

  public override IList<IEnumerator<string>>
    GetPartitions(int partitionCount)
  {
    var result = new List<IEnumerator<string>>(2);
    result.Add(CreateEnumerator(true));
    result.Add(CreateEnumerator(false));

    return result;
  }

  IEnumerator<string> CreateEnumerator(bool isEven)
  {
    foreach (var d in _data)
    {
      if (!(d.Length % 2 == 0 ^ isEven))
      yield return d;
    }
  }
}
```

4. 在 Main 方法中加入以下代码片段：

```
var partitioner = new StringPartitioner(GetTypes());
var parallelQuery = from t in partitioner.AsParallel()
  select EmulateProcessing(t);

parallelQuery.ForAll(PrintInfo);
```

5. 运行程序。

7.6.3 工作原理

为了演示我们能够为 PLINQ 查询选择自定义的分区策略，我们创建了一个非常简单的分区器，用于并行地处理奇数长度与偶数长度的字符串。为了实现这个区分器，我们从

Partitioner<T> 标准基类派生出 StringPartitioner 类来，并使用 string 作为类型参数。

我们声明只支持静态分区，方式是重载 SupportsDynamicPartitions 属性并设置其值为 false。这意味着我们预定义了分区策略。这是一个简易方法来对初始集合进行分区，但是有可能效率较低，主要取决于集合中是什么样的数据。例如，在本例中有很多奇数长度的字符串，但只有一个字符串是偶数长度，其中一个线程会早早的完成而不会帮助处理奇数长度的字符串。而动态分区意味着我们实时对初始集合进行分区，并在工作线程间平衡工作负载。

然后我们实现了 GetPartitions 方法，在其中定义了两个迭代。第一个从源集合中返回奇数长度的字符串，第二个则返回偶数长度的字符串。最后，创建了一个分区器的实例并使用它来执行一个 PLINQ 查询。我们可以看到不同的线程会处理奇数长度和偶数长度的字符串。

7.7　为 PLINQ 查询创建一个自定义的聚合器

本节展示了如何为 PLINQ 查询实现一个自定义的聚合功能。

7.7.1　准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter7\Recipe6 目录中。

7.7.2　实现方式

请执行以下步骤来了解如何为 PLINQ 查询自定义聚合功能：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
```

3. 在 Main 方法下面加入以下代码片段：

```
static ConcurrentDictionary<char, int>
  AccumulateLettersInformation(ConcurrentDictionary<char,
    int> taskTotal , string item)
{
  foreach (var c in item)
  {
    if (taskTotal.ContainsKey(c))
    {
      taskTotal[c] = taskTotal[c] + 1;
```

```
    }
    else
    {
      taskTotal[c] = 1;
    }
  }
  Console.WriteLine("{0} type was aggregated on a thread id
    {1}",item, Thread.CurrentThread.ManagedThreadId);
  return taskTotal;
}

static ConcurrentDictionary<char, int>
  MergeAccumulators(ConcurrentDictionary<char, int> total,
  ConcurrentDictionary<char, int> taskTotal)
{
  foreach (var key in taskTotal.Keys)
  {
    if (total.ContainsKey(key))
    {
      total[key] = total[key] + taskTotal[key];
    }
    else
    {
      total[key] = taskTotal[key];
    }
  }
  Console.WriteLine("---");
  Console.WriteLine("Total aggregate value was calculated
    on a thread id {0}",
    Thread.CurrentThread.ManagedThreadId);
  return total;
}

static IEnumerable<string> GetTypes()
{
  var types = AppDomain.CurrentDomain
    .GetAssemblies()
    .SelectMany(a => a.GetExportedTypes());

  return from type in types
    where type.Name.StartsWith("Web")
    select type.Name;
}
```

4. 在 Main 方法中加入以下代码片段：

```
var parallelQuery = from t in GetTypes().AsParallel()
  select t;

var parallelAggregator = parallelQuery.Aggregate(
  () => new ConcurrentDictionary<char, int>(),
  (taskTotal, item) => AccumulateLettersInformation
```

```
    (taskTotal, item),
    (total, taskTotal) => MergeAccumulators(total,
    taskTotal), total => total);

Console.WriteLine();
Console.WriteLine("There were the following letters in type
  names:");
var orderedKeys = from k in parallelAggregator.Keys
  orderby parallelAggregator[k] descending
  select k;

foreach (var c in orderedKeys)
{
  Console.WriteLine("Letter '{0}' ---- {1} times", c,
    parallelAggregator[c]);
}
```

5. 运行程序。

7.7.3　工作原理

这里我们实现了一个可以工作于 PLINQ 查询的自定义聚合机制。我们需要知道由于一个查询会被多个任务同时以并行的方式处理，那么就需要提供一种机制来并行地聚合每个任务的结果，然后将这些聚合的值合并到单个结果值中。

本节中，我们编写了一个聚合函数用来在 PLINQ 查询中统计字母数量，其返回一个 IEnumerable<string> 集合。它统计了每个集合项中的所有字母数。为了演示并行聚合过程，我们打印出了具体线程处理该聚合器的所有部分的信息。

我们使用了定义在 ParallelEnumerable 类中的 Aggregate 扩展方法来聚合 PLINQ 查询结果。它接受 4 个参数，每个都是一个函数用于执行该聚合过程的不同部分。第一个是一个工厂类，其构造了该聚合器的空的初始值。该值也被称为种子值。

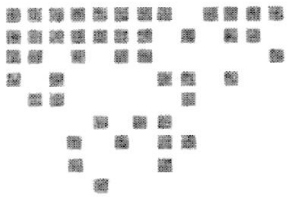


请注意提供给 Aggregate 方法的第一个值并不是聚合器功能的初始种子值，而是一个构造了该初始种子值的工厂方法。如果你只提供一个实例，其将被使用在并行运行的所有分区中，这将导致不正确的结果。

第二个函数将每个集合项聚合到分区聚合对象中。我们使用 AccumulateLettersInformation 方法来实现该函数。它迭代字符串并统计其中的字符数。这里的聚合对象在并行运行的每个查询分区中是不一样的，这就是为什么我称其为 taskTotal。

第三个函数是一个高阶聚合函数，用于从分区中将一个聚合对象合并到全局聚合对象中。我们使用 MergeAccumulators 方法来实现它。最后一个函数是一个选择器函数，指定了全局聚合对象中我们需要的确切数据。

最后，我们打印出了聚合器值，并按照在集合项中字符使用频率从高到低排序。



Chapter 8 第 8 章

使用 Reactive Extensions

本章中我们将学习另一个有意思的 .NET 库来帮助我们创建异步程序，即反应扩展（Reactive Extensions，简称 Rx）。你将学习以下小节：

- 将普通集合转换为异步的可观察集合
- 编写自定义的可观察对象
- 使用 Subject
- 创建可观察的对象
- 对可观察的集合使用 LINQ 查询
- 使用 Rx 创建异步操作

8.1 简介

之前我们已经学习过了，有好几种方式在 .NET 和 C# 中创建异步程序。其中一个是基于事件的异步模式，在之前的章节中已经提及过。引入事件的初始目的是简化**观察者**设计模式的实现。该模式常用于实现对象间的通讯。

当我们讨论任务并行库时，我们注意到事件的主要缺点是它们不能有效地相互结合。另一个缺点是基于事件的异步模式不能处理通知的顺序。想象一下我们有 IEnumerable<string> 提供给我们字符串值。然而当迭代它时，不知道每个迭代会花费多长时间。如果使用常规的 foreach 或其他的同步迭代构造方式，我们将阻塞线程直到得到下一个值，这可能会导致整个处理非常慢。这种作为客户端从生产者那里拉取值的场景被称为**基于拉取**（pull-based）的方式。

相反的方式是**基于推送**（push-based）的方式，即生产者通知客户端有新值要处理。这将把工作推给生产者，而客户端在等待另一个值的时候可以做些其他事情。因此，目标是实现类似于 IEnumerable 的异步版本的一个机制，可以生产一组序列值并按顺序通知消费者处理这些值，直到序列处理完成或抛出异常。

.NET Framework 从 4.0 版本开始包含了接口 IObservable<out T> 和 IObserver<in T> 的定义，它们一起代表了异步的基于推送的集合及其客户端。它们都来自叫做 Reactive Extensions（简称 Rx）的库，其由微软创建，用于使用可观察的集合来有效地构造事件序列，以及实际上任何其他类型的异步程序。这些接口包括在 .Net Framework 中，但这些接口的实现类以及所有其他机制仍单独的分布在 Rx 库中。



Reactive Extensions 起初是一个跨平台的库。.NET3.5，Silverlight 以及 Windows Phone 都有相应的库。甚至 JavaScript、Ruby 和 Python 都可以使用 Reactive Extensions。它也是开源的，你可以在 CodePlex 网站找到针对 .NET 的 Reactive Extensions 源码，也可以在 GitHub 找到其他实现。

最迷人的事情是可观察的集合与 LINQ 是兼容的。因此我们可以使用声明式查询以异步的方式来转换和组合这些集合。也可以使用扩展方法给 Rx 程序添加功能，从而具备通常的 LINQ 查询的同样的功能。Rx 程序也支持从所有异步编程模式（包括异步编程模型，基于事件的异步模式，以及任务并行库）转换到可观察的集合，并且支持以 Rx 内置方式运行异步操作，该方式与 TPL 很相似。

Reactive Extensions 库是一个非常强大和复杂的工具，值得单独写本书。在本章中我将展示最常用的场景，即如何高效地与异步事件序列工作。我们将了解 Reactive Extensions 框架中的关键类型，学习以不同的方式创建和维护序列。最后，将展示如何使用 Reactive Extensions 来运行异步操作并管理操作选项。

8.2　将普通集合转换为异步的可观察集合

本节将展示如何使用 Enumerable 类创建可观察的集合，并且异步处理该集合。

8.2.1　准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter1\Recipe8 目录中。

8.2.2　实现方式

请执行以下步骤来了解如何使用 Enumerable 类创建可观察的集合，以及如何异步处理

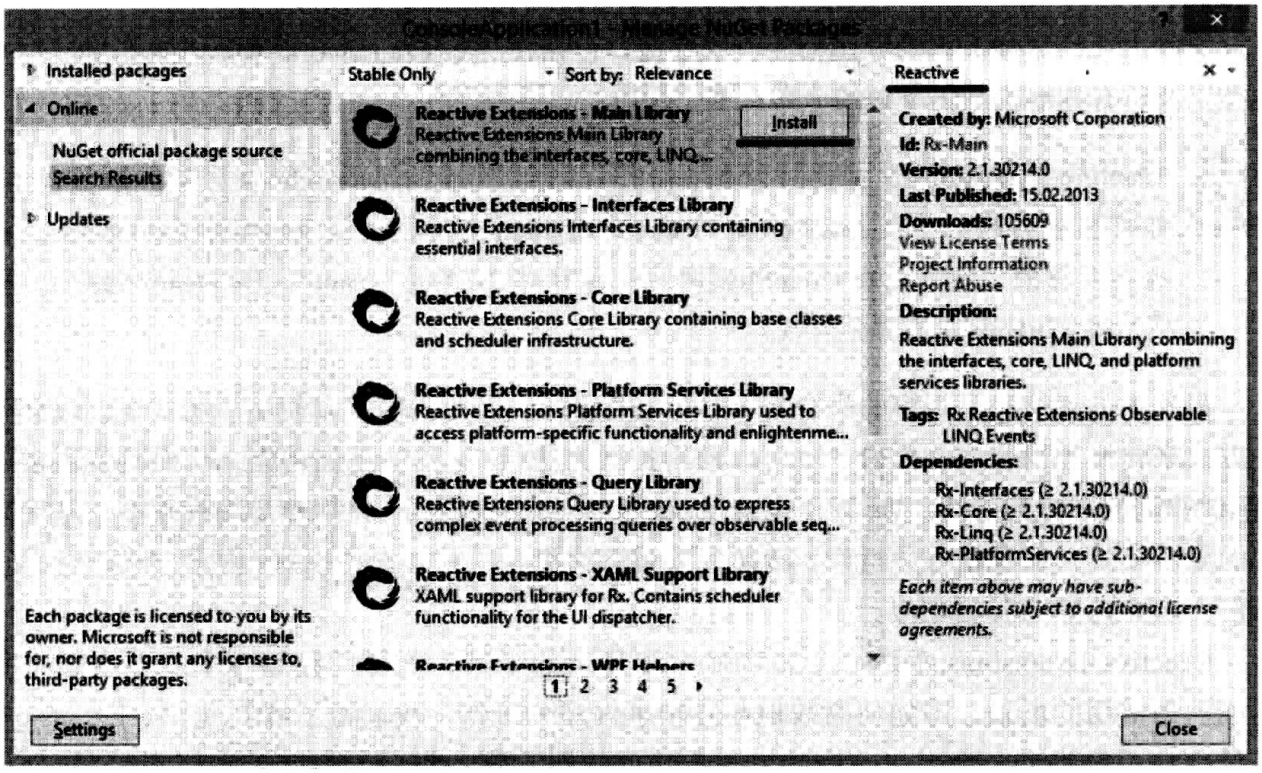
该集合：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 添加对 Reactive Extensions Main Library NuGet 包的引用。

（1）右键点击项目中引用目录并选择管理 NuGet 包菜单项。

（2）添加对 Reactive Extensions-Main Library NuGet 包的引用。你可以使用管理 NuGet 包对话框中的搜索功能，如下截图所示：



3. 在 Program.cs 文件中加入以下代码片段：

```
using System;
using System.Collections.Generic;
using System.Reactive.Concurrency;
using System.Reactive.Linq;
using System.Threading;
```

4. 在 Main 方法下面加入以下代码片段：

```
static IEnumerable<int> EnumerableEventSequence()
{
  for (int i = 0; i < 10; i++)
  {
    Thread.Sleep(TimeSpan.FromSeconds(0.5));
    yield return i;
  }
}
```

5. 在 Main 方法中加入以下代码片段：

```
foreach (int i in EnumerableEventSequence())
{
  Console.Write(i);
}
Console.WriteLine();
Console.WriteLine("IEnumerable");

IObservable<int> o =
  EnumerableEventSequence().ToObservable();
using (IDisposable subscription =
  o.Subscribe(Console.Write))
{
  Console.WriteLine();
  Console.WriteLine("IObservable");
}

o = EnumerableEventSequence().ToObservable()
  .SubscribeOn(TaskPoolScheduler.Default);
using (IDisposable subscription =
  o.Subscribe(Console.Write))
{
  Console.WriteLine();
  Console.WriteLine("IObservable async");
  Console.ReadLine();
}
```

6. 运行程序。

8.2.3　工作原理

我们使用 EnumerableEventSequence 方法模拟了一个效率不高的可枚举的集合。然后使用常用的 foreach 循环来迭代它，可以看到这一如既往的慢。等待直到所有迭代完成。

然后借助于 Reactive Extensions 库中的 ToObservable 扩展方法把可枚举的集合转换为可观察的集合。接下来订阅该可观察集合的更新，提供 Console.Write 方法作为操作，其将在每次更新该集合时被执行。结果我们得到了与上一个例子完全一样的行为。我们会等待所有迭代完成，因为我们使用了主线程来订阅这些更新。



我们使用 using 声明来封装这些订阅对象。尽管不总是必要的，但是及时消除这些订阅是一个好的实践，因为可以避免生命周期相关的错误。

为了异步化该程序，我们使用 SubscribeOn 方法并提供其 TPL 任务池调度程序。该调度程序将把订阅信息放置到 TPL 任务池中，卸除主线程的任务。这可以使 UI 在集合更新时仍保持响应并做些其他事情。为了检查该行为，你可以从代码中除移最后的 Console.ReadLine

调用。这样主线程会立即完成，从而强制所有的后台线程（包括 TPL 任务池工作线程）也一起结束，我们将不会从异步集合中得到任何输出。

如果使用了任何 UI 框架，就只能从 UI 线程与 UI 控制器进行交互。为了实现该点，我们应该对相应的调度程序使用 ObserveOn 方法。对于 Windows Presentation Foundation，我们拥有定义在名为 Rx-XAML（或 Reactive Extensions XAML 支持库）的单独的 NuGet 包中的 DispatcherScheduler 类和 ObserveOnDispatcher 扩展方法。其他的平台也有相应的单独的 NuGet 包。

8.3 编写自定义的可观察对象

本节描述了如何实现 IObservable<in T> 和 IObserver<out T> 接口来获取自定义的可观察的序列，以及如何正确地使用该序列。

8.3.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter8\Recipe2 目录中。

8.3.2 实现方式

请执行以下步骤来了解如何实现 IObservable<in T> 和 IObserver<out T> 接口来获取自定义可观察的序列，并正确地使用它：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 添加对 Reactive Extensions Main Library NuGet 包的引用。请参见 8.2 节来了解细节。

3. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Collections.Generic;
using System.Reactive.Concurrency;
using System.Reactive.Disposables;
using System.Reactive.Linq;
using System.Threading;
```

4. 在 Main 方法下面加入以下代码片段：

```
class CustomObserver : IObserver<int>
{
  public void OnNext(int value)
  {
    Console.WriteLine("Next value: {0}; Thread Id: {1}",
      value, Thread.CurrentThread.ManagedThreadId);
```

```
    }

    public void OnError(Exception error)
    {
      Console.WriteLine("Error: {0}", error.Message);
    }

    public void OnCompleted()
    {
      Console.WriteLine("Completed");
    }
}

class CustomSequence : IObservable<int>
{
    private readonly IEnumerable<int> _numbers;

    public CustomSequence(IEnumerable<int> numbers)
    {
      _numbers = numbers;
    }
    public IDisposable Subscribe(IObserver<int> observer)
    {
      foreach (var number in _numbers)
      {
        observer.OnNext(number);
      }
      observer.OnCompleted();
      return Disposable.Empty;
    }
}
```

5. 在 Main 方法中加入以下代码片段：

```
var observer = new CustomObserver();

var goodObservable = new CustomSequence(new[] {1, 2, 3, 4,
  5});
var badObservable = new CustomSequence(null);

using (IDisposable subscription =
  goodObservable.Subscribe(observer))
{
}

using (IDisposable subscription =
  goodObservable.SubscribeOn(TaskPoolScheduler.Default)
  .Subscribe(observer))
{
  Thread.Sleep(100);
}

using (IDisposable subscription =
```

```
    badObservable.SubscribeOn(TaskPoolScheduler.Default)
    .Subscribe(observer))
{
    Console.ReadLine();
}
```

6. 运行程序。

8.3.3 工作原理

这里我们先实现了定义的观察者，简单地向控制台打印可观察集合中下一项的相关信息、错误信息以及序列完成信息。这是非常简单的消费者代码，没什么特别的。

有意思的部分是可观察的集合实现。我们在构造函数中接受一个数字枚举，故意不检查它是否为 null。然后创建一个订阅的观察者，我们迭代该集合并通知观察者枚举集合中的每个项。

然后我们演示了该实际订阅行为。正如我们所看到的，通过 SubscribeOn 方法实现了异步。该方法是 IObservable 的一个扩展方法，并且包含异步订阅逻辑。我们不关心如何实现可观察的集合中的异步，而是用 Reactive Extensions 库中的标准实现。

当订阅平常的可观察集合时，我们只是从中得到了其所有元素。现在它是异步的，所以需要等待一些时间来让异步操作完成，然后打印信息并等待用户输入。

最后我们尝试订阅下一个可观察的集合，我们迭代一个空的枚举因此得到了一个空引用异常。可以看到该异常被正确的处理，并且 OnError 方法被执行，打印出了错误细节。

8.4 使用 Subject

本节展示了如何使用 Reactive Extensions 库中的 Subject 类型家族。

8.4.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter8\Recipe3 目录中。

8.4.2 实现方式

请执行以下步骤来了解如何使用 Reactive Extensions 库中的 Subject 类型家族：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。
2. 添加对 Reactive Extensions Main Library NuGet 包的引用。请参见 8.2 节来了解细节。
3. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Reactive.Subjects;
using System.Threading;
```

4. 在 Main 方法下面加入以下代码片段：

```
static IDisposable OutputToConsole<T>(IObservable<T> sequence)
{
  return sequence.Subscribe(
    obj => Console.WriteLine("{0}", obj)
    , ex => Console.WriteLine("Error: {0}", ex.Message)
    , () => Console.WriteLine("Completed")
  );
}
```

5. 在 Main 方法中加入以下代码片段：

```
Console.WriteLine("Subject");
var subject = new Subject<string>();

subject.OnNext("A");
using (var subscription = OutputToConsole(subject))
{
  subject.OnNext("B");
  subject.OnNext("C");
  subject.OnNext("D");
  subject.OnCompleted();
  subject.OnNext("Will not be printed out");
}

Console.WriteLine("ReplaySubject");
var replaySubject = new ReplaySubject<string>();

replaySubject.OnNext("A");
using (var subscription = OutputToConsole(replaySubject))
{
  replaySubject.OnNext("B");
  replaySubject.OnNext("C");
  replaySubject.OnNext("D");
  replaySubject.OnCompleted();
}

Console.WriteLine("Buffered ReplaySubject");
var bufferedSubject = new ReplaySubject<string>(2);

bufferedSubject.OnNext("A");
bufferedSubject.OnNext("B");
bufferedSubject.OnNext("C");
using (var subscription = OutputToConsole(bufferedSubject))
{
  bufferedSubject.OnNext("D");
```

```
  bufferedSubject.OnCompleted();
}

Console.WriteLine("Time window ReplaySubject");
var timeSubject = new ReplaySubject<string>
  (TimeSpan.FromMilliseconds(200));

timeSubject.OnNext("A");
Thread.Sleep(TimeSpan.FromMilliseconds(100));

timeSubject.OnNext("B");
Thread.Sleep(TimeSpan.FromMilliseconds(100));
timeSubject.OnNext("C");
Thread.Sleep(TimeSpan.FromMilliseconds(100));
using (var subscription = OutputToConsole(timeSubject))
{
  Thread.Sleep(TimeSpan.FromMilliseconds(300));
  timeSubject.OnNext("D");
  timeSubject.OnCompleted();
}

Console.WriteLine("AsyncSubject");
var asyncSubject = new AsyncSubject<string>();

asyncSubject.OnNext("A");
using (var subscription = OutputToConsole(asyncSubject))
{
  asyncSubject.OnNext("B");
  asyncSubject.OnNext("C");
  asyncSubject.OnNext("D");
  asyncSubject.OnCompleted();
}

Console.WriteLine("BehaviorSubject");
var behaviorSubject = new BehaviorSubject<string>
  ("Default");
using (var subscription = OutputToConsole(behaviorSubject))
{
  behaviorSubject.OnNext("B");
  behaviorSubject.OnNext("C");
  behaviorSubject.OnNext("D");
  behaviorSubject.OnCompleted();
}
```

6. 运行程序。

8.4.3 工作原理

在本程序中我们遍历了 Subject 类型家族的不同变种。Subject 代表了 IObservable 和 IObserver 这两个接口的实现。在不同的代理场景中它非常有用，比如将事件从多个源中转换

到一个流中，反过来也一样，广播事件序列给多个订阅者。Subject 与 Reactive Extensions 一起使用也非常方便。

先从基本的 Subject 类型开始。一旦订阅了 Subject，它就会把事件序列发送给订阅者。在本例中，字符串 A 不会被打印，因为订阅发生在传播之后。基于此，当对 Observable 调用 OnCompleted 和 OnError 方法时，它停止了进一步的事件序列传播，所以最后的字符串也不能打印出来。

接下来的类型 ReplaySubject 是非常灵活的，允许我们实现三个附加的场景。首先，从广播开始可以缓存所有的事件，如果稍后订阅，将先获得之前所有的事件。第二个例子演示该行为。这里控制台输出了 4 个字符串，因为第一个事件将被缓存并传送给稍后的订阅者。

接着我们指定了 ReplaySubject 的缓存大小和时间窗口大小。接下来的例子中，我们设置该 subject 的缓存只能容纳 2 个事件。如果广播了更多的事件，只有最后两个会被再次传送给订阅者。所以我们看不到第一个字符串，因为当订阅它时只有 B 和 C 在 subject 的缓存中。使用时间窗口的行为也类似。我们可以指定 subject 只缓存发生在确定时间以内的事件，丢弃掉旧数据。因此，第四个例子中，我们只看到了最后两个事件。较老的事件并不符合时间窗口限制。

AsyncSubject 类似于任务并行库中的 Task 类型。它代表了单个异步操作。如果有多个事件发布，它将等待事件序列完成，并把最后一个事件提供给订阅者。

BehaviorSubject 与 ReplaySubject 类型很相似，但它只缓存一个值，并允许万一还没有发送任何通知时，指定一个默认值。在最后一个例子中，可以看到打印出了所有字符串，因为我们提供了默认值，所有其他事件都已经发生在订阅之后。如果向上移动 behaviorSubject.OnNext("B")；代码行到 Default 事件下面，其将替换掉输出中的默认值。

8.5　创建可观察的对象

本节描述了创建可观察的对象的多种方式。

8.5.1　准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter8\Recipe4 目录中。

8.5.2　实现方式

请执行以下步骤来了解创建可观察的对象的不同方式：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 添加对 Reactive Extensions Main Library NuGet 包的引用。请参见 8.2 节来了解细节。

3. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Reactive.Disposables;
using System.Reactive.Linq;
using System.Threading;
```

4. 在 Main 方法下面加入以下代码片段：

```
static IDisposable OutputToConsole<T>(IObservable<T> sequence)
{
  return sequence.Subscribe(
    obj => Console.WriteLine("{0}", obj)
    , ex => Console.WriteLine("Error: {0}", ex.Message)
    , () => Console.WriteLine("Completed")
  );
}
```

5. 在 Main 方法中加入以下代码片段：

```
IObservable<int> o = Observable.Return(0);
using (var sub = OutputToConsole(o));
Console.WriteLine(" ---------------- ");

o = Observable.Empty<int>();
using (var sub = OutputToConsole(o));
Console.WriteLine(" ---------------- ");

o = Observable.Throw<int>(new Exception());
using (var sub = OutputToConsole(o));
Console.WriteLine(" ---------------- ");

o = Observable.Repeat(42);
using (var sub = OutputToConsole(o.Take(5)));
Console.WriteLine(" ---------------- ");

o = Observable.Range(0, 10);
using (var sub = OutputToConsole(o));
Console.WriteLine(" ---------------- ");

o = Observable.Create<int>(ob => {
  for (int i = 0; i < 10; i++)
  {
    ob.OnNext(i);
  }
  return Disposable.Empty;
});
using (var sub = OutputToConsole(o)) ;
Console.WriteLine(" ---------------- ");
```

```
o = Observable.Generate(
  0 // initial state
  , i => i < 5 // while this is true we continue the
  sequence
  , i => ++i // iteration
  , i => i*2 // selecting result
);
using (var sub = OutputToConsole(o));
Console.WriteLine(" ---------------- ");

IObservable<long> ol =
  Observable.Interval(TimeSpan.FromSeconds(1));
using (var sub = OutputToConsole(ol))
{
  Thread.Sleep(TimeSpan.FromSeconds(3));
};
Console.WriteLine(" ---------------- ");

ol = Observable.Timer(DateTimeOffset.Now.AddSeconds(2));
using (var sub = OutputToConsole(ol))
{
  Thread.Sleep(TimeSpan.FromSeconds(3));
};
Console.WriteLine(" ---------------- ");
```

6. 运行程序。

8.5.3　工作原理

这里我们展示了创建可观察的对象的不同场景。大部分的功能是通过 Observable 类型的静态工厂方法实现的。前两个例子展示了如何使用值或者不使用值来创建一个 Observable 方法。接下来的例子中使用 Observable.Throw 来构造一个 Observable 类，从而触发其观察者的 OnError 处理器。

Observable.Repeat 方法代表了一个无尽的序列。该方法有多个重载。这里我们通过重复值 42 构造了一个无尽的序列。然后使用 LINQ 的 Take 方法从该序列中提取 5 个元素。Observable. Range 表示了一组值，其与 Enumerable.Range 非常类似。

Observable.Create 方法支持更多的自定义场景。有相当多的重载允许我们使用取消标志和任务。但我们先看看最简单的重载。它接受一个函数，该函数接受一个观察者实例，并且返回 IDisposable 对象来代表订阅者。如果需要清除任何资源，可以在此放置清除逻辑，但本例中只返回一个空的 disposable，因为实际上并不需要。

Observable.Generate 是另一个创建自定义序列的方式。我们必须为序列提供一个初始值，然后提供一个断言，来决定是否需要生成更多元素或者完成序列。接着提供了一个迭代逻辑，在本例中递增计数器值。最后一个参数是一个选择器函数，允许我们定制化结果。

最后两个方法处理计时器。Observable.Interval 会以 TimeSpan 间隔产生计时器标记事件，Observable.Timer 也指定了启动时间。

8.6 对可观察的集合使用 LINQ 查询

本节展示了如何使用 LINQ 来查询异步事件序列。

8.6.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter8\Recipe5 目录中。

8.6.2 实现方式

请执行以下步骤来了解如何对可观察的集合使用 LINQ 查询：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 添加对 Reactive Extensions Main Library NuGet 包的引用。请参见 8.2 节来了解细节。

3. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Reactive.Linq;
```

4. 在 Main 方法下面加入以下代码片段：

```
static IDisposable OutputToConsole<T>(IObservable<T> sequence,
int innerLevel)
{
  string delimiter = innerLevel == 0 ? string.Empty : new
    string('-', innerLevel*3);
  return sequence.Subscribe(
    obj => Console.WriteLine("{0}{1}", delimiter, obj)
    , ex => Console.WriteLine("Error: {0}", ex.Message)
    , () => Console.WriteLine("{0}Completed", delimiter)
  );
}
```

5. 在 Main 方法中加入以下代码片段：

```
IObservable<long> sequence =
  Observable.Interval(TimeSpan.FromMilliseconds(50))
  .Take(21);

var evenNumbers = from n in sequence
  where n % 2 == 0
  select n;

var oddNumbers = from n in sequence
```

```
    where n % 2 != 0
    select n;

var combine = from n in evenNumbers.Concat(oddNumbers)
    select n;

var nums = (from n in combine
    where n % 5 == 0
    select n)
    .Do(n => Console.WriteLine("------Number {0} is processed
    in Do method", n));

using (var sub = OutputToConsole(sequence, 0))
using (var sub2 = OutputToConsole(combine, 1))
using (var sub3 = OutputToConsole(nums, 2))
{
    Console.WriteLine("Press enter to finish the demo");
    Console.ReadLine();
}
```

6. 运行程序。

8.6.3　工作原理

对可观察的事件序列使用 LINQ 的能力是 Reactive Extensions 框架的主要优势。其实有很多不同的使用场景。遗憾的是在这里无法展示全部。我尝试提供一个非常简单的说明示例，这里没有非常复杂的细节，展示了 LINQ 查询应用到异步的可观察的集合的精华。

首先我们创建了 Observable 事件来创建一个数字序列，每个数字用时 50 毫秒。我们从初始值 0 开始，产生 21 个这样的事件。然后将 LINQ 查询结合到该序列中。首先，我们只选择序列中的奇数，然后再只选择偶数，最后再连接这两个序列。

最后的查询展示了如何使用非常有用的方法 Do，其允许引入副作用（side effect）。比如，从结果序列中记录每个值。为了运行所有的查询，我们创建了嵌套的订阅。由于该序列初始时是异步的，所以要非常小心订阅的生命周期。外部范围代表了向计时器的订阅，内部订阅则各自处理组合的序列查询和副作用查询。如果按下回车键太早，我们只是停止对计时器的订阅，这会导致停止该示例运行。

当运行该示例时，我们可以看到不同的查询实时交互的实际过程。可以看到这些查询是惰性的，只有我们订阅了其结果，查询才会运行。计时器事件序列打印在第一列。当事件号查询得到一个事件号时，也会打印出该号，并使用 --- 做前缀来区分第一列的序列结果。最终查询结果打印在右边的列。

当程序运行时，可以看到计时器序列、奇数序列和副作用序列并行的运行。只有连接操作会等待直到奇数序列完成。如果我们不连接这些序列，我们将有 4 个并行序列事件以

最高效的方式相互交互！这展示了 Reactive Extensions 的真正力量，是深入学习该库的好的开端。

8.7 使用 Rx 创建异步操作

本节展示了如何为定义在其他编程模式中的异步操作创建 Observable 对象。

8.7.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter8\Recipe6 目录中。

8.7.2 实现方式

请执行以下步骤来了解如何创建使用 Rx 创建异步操作：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 添加对 Reactive Extensions Main Library NuGet 包的引用。请参见 8.2 节来了解细节。

3. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Reactive;
using System.Reactive.Linq;
using System.Reactive.Threading.Tasks;
using System.Threading;
using System.Threading.Tasks;
using System.Timers;
using Timer = System.Timers.Timer;
```

4. 在 Main 方法下面加入以下代码片段：

```
static async Task<T> AwaitOnObservable<T>(IObservable<T>
  observable)
{
  T obj = await observable;
  Console.WriteLine("{0}", obj );
  return obj;
}

static Task<string> LongRunningOperationTaskAsync(string
  name)
{
  return Task.Run(() => LongRunningOperation(name));
}
```

```
static IObservable<string> LongRunningOperationAsync(string
  name)
{
  return Observable.Start(() =>
    LongRunningOperation(name));
}

static string LongRunningOperation(string name)
{
  Thread.Sleep(TimeSpan.FromSeconds(1));
  return string.Format("Task {0} is completed. Thread Id
    {1}", name, Thread.CurrentThread.ManagedThreadId);
}

static IDisposable OutputToConsole
  (IObservable<EventPattern<ElapsedEventArgs>> sequence)
{
  return sequence.Subscribe(
    obj => Console.WriteLine("{0}",
    obj.EventArgs.SignalTime)
    , ex => Console.WriteLine("Error: {0}", ex.Message)
    , () => Console.WriteLine("Completed")
  );
}

static IDisposable OutputToConsole<T>(IObservable<T>
  sequence)
{
  return sequence.Subscribe(
    obj => Console.WriteLine("{0}", obj)
    , ex => Console.WriteLine("Error: {0}", ex.Message)
    , () => Console.WriteLine("Completed")
  );
}
```

5. 在 Main 方法中加入以下代码片段：

```
IObservable<string> o = LongRunningOperationAsync("Task1");
using (var sub = OutputToConsole(o))
{
  Thread.Sleep(TimeSpan.FromSeconds(2));
};
Console.WriteLine(" ---------------- ");
Task<string> t = LongRunningOperationTaskAsync("Task2");
using (var sub = OutputToConsole(t.ToObservable()))
{
  Thread.Sleep(TimeSpan.FromSeconds(2));
};
Console.WriteLine(" ---------------- ");

AsyncDelegate asyncMethod = LongRunningOperation;

// marked as obsolete, use tasks instead
```

```
Func<string, IObservable<string>> observableFactory =
  Observable.FromAsyncPattern<string, string>(
  asyncMethod.BeginInvoke, asyncMethod.EndInvoke);
o = observableFactory("Task3");
using (var sub = OutputToConsole(o))
{
  Thread.Sleep(TimeSpan.FromSeconds(2));
};
Console.WriteLine(" ---------------- ");

o = observableFactory("Task4");
AwaitOnObservable(o).Wait();
Console.WriteLine(" ---------------- ");

using (var timer = new Timer(1000))
{
  var ot = Observable.FromEventPattern<ElapsedEventHandler,
    ElapsedEventArgs>(
    h => timer.Elapsed += h
    ,h => timer.Elapsed -= h);
  timer.Start();

  using (var sub = OutputToConsole(ot))
  {
    Thread.Sleep(TimeSpan.FromSeconds(5));
  }
  Console.WriteLine(" ---------------- ");
  timer.Stop();
}
```

6. 运行程序。

8.7.3 工作原理

本节展示了如何将不同类型的异步操作转换为 Observable 类。步骤 5 的第一个代码片段使用了 Observable.Start 方法，其与 TPL 中的 Task.Run 方法很相似。它启动了异步操作并返回一个字符串结果，然后退出。



我强烈建议使用任务并行库来执行异步操作。虽然 Reactive Extensions 也支持该功能，但是为了避免使用不当，当讲单独的异步操作时最好使用任务，而只有需要与事件序列工作时才用 Rx。另一个建议是将单独的异步操作转换为任务，只有在需要时才把任务转换为 observable 类。

然后，我们同样通过简单调用 ToObservable 扩展方法把任务转换为 Observable 方法。步骤 5 的接下来的代码片段是关于将异步编程模块模式转换为 Observable 类。通常你可能将 APM 转换为任务，再把任务转换为 Observable 类。然而其实可以直接转换，该例子演示了

如何运行一个异步委托并将其封装到 Observable 操作中。

步骤 5 的下一段代码片段展示了我们可以对 Observable 操作使用 await。我们无法给一个入口方法（例如 Main）使用 async 修饰符，所以引入了一个单独的方法来运行一个任务，并等待该任务在 Main 方法中完成执行。

步骤 5 的最后一部分代码片段与之前是一样的，但是这里我们把基于事件的异步模式直接转换为 Observable 类。我们创建了一个计时器并消费该计时器事件 5 秒钟。然后对该计时器调用 dispose 方法以清除资源。



Chapter 9 第 9 章

使用异步 I/O

在本章中，我们将学习异步输入 / 输出的操作细节。你将学到以下内容：

- 异步地使用文件
- 编写一个异步的 HTTP 服务器和客户端
- 异步地操作数据库
- 异步调用 WCF 服务

9.1 简介

之前的章节中我们已经讨论了正确地使用异步输入 / 输出操作是多么重要。为什么如此重要呢？为了加深理解，让我们考虑两种应用程序。

如果在客户端运行程序，最重要的事情之一是有一个响应的用户界面。这意味着无论应用程序发生什么，所有的用户界面元素（比如按钮和进度条）都要保持快速运行，用户能够从应用程序得到快速响应。达到该点并不容易！如果你尝试在 Windows 系统中打开记事本编辑器并加载一个有几个兆字节大小的文档，应用程序窗口将冻结一段显著的时间，因为整个文档要先从硬盘中加载，然后程序才能开始处理用户输入。

这是一个非常重要的问题，在该情况下，唯一方案是无论如何都要避免阻塞 UI 线程。这反过来意味着为了防止阻塞 UI 线程，每个与 UI 有关的 API 必须只被允许异步调用。这是 Window 8 操作系统重新升级 API 的关键原因，其几乎把每个方法替换为异步方式。但是如

果应用程序使用多线程来达到此目的会影响性能吗？当然会！然而考虑到只有一个用户，那么这是划算的。如果应用程序可以使用电脑的所有能力从而变得更加高效，而且该能力只为运行程序的唯一用户服务，这是好事。

接下来看看第二种情况。如果程序运行在服务器端，则是完全不同的情形。可伸缩性是最高优先级，这意味着单个用户消耗越少的资源越好。如果为每个用户创建多个线程，则可伸缩性并不好。以高效的方式来平衡应用程序资源的消耗是个非常复杂的问题。例如，在 ASP.NET（其是微软提供的 web 应用程序平台）中，我们使用工作线程池来服务客户端请求。该池的工作线程数是有限的，所以不得不最小化每个工作线程的使用时间以便达到高伸缩性。这意味着需要把工作线程越快越好地放回到池中，从而可以服务下一个请求。如果我们启动了一个需要计算的异步操作，则整个工作流程会很低效。首先从线程池中取出一个工作线程用以服务客户端请求。然后取出另一个工作线程并开始处理异步操作。现在有两个工作线程都在处理请求，如果第一个线程能做些有用的事则非常好！遗憾的是，通常情况是我们简单等待异步操作完成，但是我们却消费了两个工作线程，而不是一个。在该场景中，异步比同步执行实际上更糟糕！我们不需要使用所有 CPU 核心，因为我们已经在服务很多客户端，它们已经使用了 CP 的所有计算能力。我们无须保持第一个线程响应，因为这没有用户界面。那么为什么我们应该在服务器端使用异步呢？

答案是只有异步输入 / 输出操作才应该使用异步。目前，现代计算机通常有一个磁盘驱动器来存储文件，一块网卡来通过网络发送与接收数据。所有这些设备都有自己的微型计算机，以非常底层的方式来管理输入 / 输出操作并发信号给操作系统结果。这又是一个非常复杂的主题。但为了让概念清楚，我们可以这样说，有一种方式让程序员开始一个输入 / 输出操作，并提供给操作系统一段代码，当操作完成后被该代码会被调用。在启动 I/O 任务与完成之间，并不需要 CPU 工作。这是由相应的磁盘和网络控制器的微型计算机完成的。这种执行 I/O 任务的方式被称为 I/O 线程。实现时使用的是 .NET 线程池，并且使用了一个来自操作系统的基础设施，叫做 I/O 完成端口。

在 APS.NET 中，一旦有一个异步的 I/O 操作在工作线程中开始时，它会被立即返回到线程池中。当该操作继续运行时，该线程可以服务其他的客户端。最终，当操作发出信号完成时，ASP.NET 基础设施从线程池中获取一个空闲的工作线程（该线程可能与操作开始时的线程不同），然后会完成该操作。

好的，我们现在了解了 I/O 线程对服务器应用程序的重要性。遗憾的是，很难看出哪些 API 在底层使用了 I/O 线程。除了学习源代码外，唯一的方式是简单知道哪个 .NET Framework 类库对 I/O 线程进行了优化。在本章中，我们将学习如何使用一些这样的 API。我们将学习如何异步操作文件，如何使用网络 I/O 来创建一个 HTTP 服务器并调用 Windows Communication Foundation 服务，以及如何使用异步 API 来查询数据库。



另一个需要考虑的重要问题是并行。由于一些原因，集中地并行磁盘操作可能导致很低的性能。请记住并行 I/O 操作经常非常低效，顺序执行 I/O 要好一些，但是要以异步的方式执行。

9.2 异步地使用文件

本节讲述了如何创建一个文件，并且以异步的方式读写数据。

9.2.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter9\Recipe1 目录中。

9.2.2 实现方式

请执行以下步骤来了解如何异步地使用文件：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
const int BUFFER_SIZE = 4096;

async static Task ProcessAsynchronousIO()
{
  using (var stream = new FileStream(
    "test1.txt", FileMode.Create, FileAccess.ReadWrite,
    FileShare.None, BUFFER_SIZE))
  {
    Console.WriteLine("1. Uses I/O Threads: {0}",
      stream.IsAsync);

    byte[] buffer = Encoding.UTF8.GetBytes
      (CreateFileContent());
  var writeTask = Task.Factory.FromAsync(
    stream.BeginWrite, stream.EndWrite, buffer, 0,
    buffer.Length, null);

  await writeTask;
}
```

```
using (var stream = new FileStream("test2.txt",
  FileMode.Create, FileAccess.ReadWrite,
  FileShare.None, BUFFER_SIZE, FileOptions.Asynchronous))
{
  Console.WriteLine("2. Uses I/O Threads: {0}",
    stream.IsAsync);

  byte[] buffer = Encoding.UTF8.GetBytes
    (CreateFileContent());
  var writeTask = Task.Factory.FromAsync(
    stream.BeginWrite, stream.EndWrite, buffer, 0,
    buffer.Length, null);

  await writeTask;
}

using (var stream = File.Create("test3.txt", BUFFER_SIZE,
  FileOptions.Asynchronous))
using (var sw = new StreamWriter(stream))
{
  Console.WriteLine("3. Uses I/O Threads: {0}",
    stream.IsAsync);
  await sw.WriteAsync(CreateFileContent());
}

using (var sw = new StreamWriter("test4.txt", true))
{
  Console.WriteLine("4. Uses I/O Threads: {0}",
    ((FileStream)sw.BaseStream).IsAsync);
  await sw.WriteAsync(CreateFileContent());
}

Console.WriteLine("Starting parsing files in parallel");

Task<long>[] readTasks = new Task<long>[4];
for (int i = 0; i < 4; i++)
{
  readTasks[i] = SumFileContent
    (string.Format("test{0}.txt", i + 1));
}

long[] sums = await Task.WhenAll(readTasks);
  Console.WriteLine("Sum in all files: {0}", sums.Sum());

  Console.WriteLine("Deleting files...");

  Task[] deleteTasks = new Task[4];
  for (int i = 0; i < 4; i++)
  {
    string fileName = string.Format("test{0}.txt", i + 1);
    deleteTasks[i] = SimulateAsynchronousDelete(fileName);
  }

  await Task.WhenAll(deleteTasks);

  Console.WriteLine("Deleting complete.");
}
```

```
static string CreateFileContent()
{
  var sb = new StringBuilder();
  for (int i = 0; i < 100000; i++)
  {
    sb.AppendFormat("{0}", new Random(i).Next(0, 99999));
    sb.AppendLine();
  }
  return sb.ToString();
}

async static Task<long> SumFileContent(string fileName)
{
  using (var stream = new FileStream(fileName,
    FileMode.Open, FileAccess.Read,
    FileShare.None, BUFFER_SIZE, FileOptions.Asynchronous))
  using (var sr = new StreamReader(stream))
  {
    long sum = 0;
    while (sr.Peek() > -1)
    {
      string line = await sr.ReadLineAsync();
      sum += long.Parse(line);
    }

    return sum;
  }
}

static Task SimulateAsynchronousDelete(string fileName)
{
  return Task.Run(() => File.Delete(fileName));
}
```

4. 在 Main 方法中加入以下代码片段：

```
var t = ProcessAsynchronousIO();
t.GetAwaiter().GetResult();
```

5. 运行程序。

9.2.3 工作原理

当程序运行时，我们以不同的方式创建了 4 个文件，并且填充了随机数据。在第一个例子中，使用的是 FileStream 类以及其方法，将异步编程模型 API 转换成任务。第二个例子中也一样，但是给 FileStream 构造函数提供了 FileStream.Asynchronous 参数。



使用 FileOptions.Asynchronous 选项是非常重要的。如果忽略该选项，我们依然可以以异步的方式使用该文件，但这只是在线程池中的异步委托调用。只有提供了该选项（或者在另一个构造函数重载中使用 bool useAsync），才能对 FileStream 类使用异步 I/O。

第三个例子使用了一些简化的 API，比如 File.Create 方法和 StreamWrite 类。它也使用 I/O 线程，我们可以使用 Stream.IsAsync 属性来检查。最后一个例子说明了过分简化也不好。这里我们借助于异步委托调用来模拟异步 I/O，其实并没有使用异步 I/O。

接着并行地异步地从所有文件中读取数据，统计每个文件内容，然后求总和。最后，删除所有文件。由于在任何非 Windows 商店应用程序中并没有异步删除文件的 API，我们使用 Task.Run 工厂方法来模拟异步删除文件。

9.3　编写一个异步的 HTTP 服务器和客户端

本节展示了如何编写一个简单的异步 HTTP 服务器。

9.3.1　准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter9\Recipe2 目录中。

9.3.2　实现方式

接下来的步骤演示了如何创建一个简单的异步 HTTP 服务器：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 添加对 System.Net.Http 框架库的引用。

3. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.IO;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
```

4. 在 Main 方法下面加入以下代码片段：

```
static async Task GetResponseAsync(string url)
{
  using (var client = new HttpClient())
  {
    HttpResponseMessage responseMessage = await
      client.GetAsync(url);
    string responseHeaders = responseMessage
      .Headers.ToString();
    string response = await responseMessage.Content
      .ReadAsStringAsync();

    Console.WriteLine("Response headers:");
```

```
    Console.WriteLine(responseHeaders);
    Console.WriteLine("Response body:");
    Console.WriteLine(response);
  }
}

class AsyncHttpServer
{
  readonly HttpListener _listener;
  const string RESPONSE_TEMPLATE =
    "<html><head><title>Test</title></head><body><h2>Test
    page</h2><h4>Today is: {0}</h4></body></html>";

  public AsyncHttpServer(int portNumber)
  {
    _listener = new HttpListener();
    _listener.Prefixes.Add(string.Format("http://+:{0}/",
      portNumber));
  }

  public async Task Start()
  {
    _listener.Start();

    while (true)
    {
      var ctx = await _listener.GetContextAsync();
      Console.WriteLine("Client connected...");
      string response = string.Format(RESPONSE_TEMPLATE,
        DateTime.Now);

      using (var sw = new StreamWriter
        (ctx.Response.OutputStream))
      {
        await sw.WriteAsync(response);
        await sw.FlushAsync();
      }
    }
  }

  public async Task Stop()
  {
    _listener.Abort();
  }
}
```

5. 在 Main 方法中加入以下代码片段：

```
var server = new AsyncHttpServer(portNumber: 1234);
var t = Task.Run(() => server.Start());
Console.WriteLine("Listening on port 1234. Open
  http://localhost:1234 in your browser.");
Console.WriteLine("Trying to connect:");
Console.WriteLine();
```

```
GetResponseAsync("http://localhost:1234").GetAwaiter()
  .GetResult();

Console.WriteLine();
Console.WriteLine("Press Enter to stop the server.");
Console.ReadLine();

server.Stop().GetAwaiter().GetResult();
```

6. 运行程序。

9.3.3 工作原理

这里我们通过 HttpListener 类实现了一个非常简单的 web 服务器。也使用了 TcpListener 类进行 TCP 套接字 I/O 操作。我们配置该监听器接收任何主机到本地机器 1234 端口的连接。然后在单独的工作线程中启动该监听器，从而在主线程中可以控制该监听器。

当使用 GetContextAsync 方法时会发生异步 I/O 操作。遗憾的是，其并不接收 CancellationToken 从而实现取消功能。所以如果想关闭该服务器，只需调用 _listener.Abort 方法，这将丢弃所有连接并关闭该服务器。

为了对该服务器执行一个异步请求，我们使用了统一命名空间下的 System.Net.Http 集合中的 HttpClient 类。我们使用 Get.Async 方法来发起一个异步的 HTTP GET 请求。还有其他的方法用于发起其他 HTTP 请求，比如 POST、DELETE 以及 PUT。HttpClient 还有很多其他的选项，比如使用不同的格式（比如 XML 和 JSON）来序列化和反序列化对象，指定代理服务器地址，认证以及其他配置。

当运行该程序时，可以看到该服务器被启动起来。在服务器端代码中，我们使用 GetContextAsync 方法来接收新的客户端连接。当有新的客户端连接时该方法就会返回，我们简单的输出一个包含当前日期和时间的非常基础的 HTML 作为响应。然后我们请求服务器并打印出响应头和内容。你也可以打开浏览器访问 http://localhost:1234/ 地址。你将看到相同的响应结果显示在浏览器窗口。

9.4 异步操作数据库

本节演示了创建数据库，以及异步地操作数据、读取数据的过程。

9.4.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter9\Recipe3 目录中。

9.4.2 实现方式

请执行以下步骤来了解创建数据库，以及异步地操作数据、读取数据的过程：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.IO;
using System.Reflection;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
async static Task ProcessAsynchronousIO(string dbName)
{
  try
  {
    const string connectionString = @"Data
      Source=(LocalDB)\v11.0;Initial
      Catalog=master;Integrated Security=True";
    string outputFolder = Path.GetDirectoryName
      (Assembly.GetExecutingAssembly().Location);
    string dbFileName = Path.Combine(outputFolder,
      string.Format(@".\{0}.mdf", dbName));
    string dbLogFileName = Path.Combine(outputFolder,
      string.Format(@".\{0}_log.ldf", dbName));
    string dbConnectionString = string.Format(@"Data
      Source=(LocalDB)\v11.0;AttachDBFileName={1};Initial
      Catalog={0};Integrated Security=True;", dbName,
      dbFileName);

    using (var connection = new SqlConnection
      (connectionString))
    {
      await connection.OpenAsync();

      if (File.Exists(dbFileName))
      {
        Console.WriteLine("Detaching the database...");

        var detachCommand = new SqlCommand("sp_detach_db",
          connection);
        detachCommand.CommandType = CommandType
          .StoredProcedure;
    detachCommand.Parameters.AddWithValue("@dbname",
      dbName);

    await detachCommand.ExecuteNonQueryAsync();

    Console.WriteLine("The database was detached
      successfully.");
```

```
    Console.WriteLine("Deleting the database...");

    if(File.Exists(dbLogFileName)) File.Delete
      (dbLogFileName);
    File.Delete(dbFileName);

    Console.WriteLine("The database was deleted
      successfully.");
  }

  Console.WriteLine("Creating the database...");
  string createCommand = String.Format("CREATE DATABASE
    {0} ON (NAME = N'{0}', FILENAME = '{1}')", dbName,
    dbFileName);
  var cmd = new SqlCommand(createCommand, connection);

  await cmd.ExecuteNonQueryAsync();
  Console.WriteLine("The database was created
    successfully");
}

using (var connection = new SqlConnection(
  dbConnectionString))
{
  await connection.OpenAsync();

  var cmd = new SqlCommand("SELECT newid()",
    connection);
  var result = await cmd.ExecuteScalarAsync();

  Console.WriteLine("New GUID from DataBase: {0}",
    result);

  cmd = new SqlCommand(@"CREATE TABLE
    [dbo].[CustomTable]( [ID] [int] IDENTITY(1,1) NOT
    NULL, [Name] [nvarchar](50) NOT NULL,
    CONSTRAINT [PK_ID] PRIMARY KEY CLUSTERED ([ID] ASC)
    ON [PRIMARY]) ON [PRIMARY]", connection);
  await cmd.ExecuteNonQueryAsync();

      Console.WriteLine("Table was created successfully.");

      cmd = new SqlCommand(@"INSERT INTO [dbo]
        .[CustomTable] (Name) VALUES ('John');
      INSERT INTO [dbo].[CustomTable] (Name) VALUES
        ('Peter');
      INSERT INTO [dbo].[CustomTable] (Name) VALUES
        ('James');
      INSERT INTO [dbo].[CustomTable] (Name) VALUES
        ('Eugene');", connection);
      await cmd.ExecuteNonQueryAsync();

      Console.WriteLine("Inserted data successfully  ");
      Console.WriteLine("Reading data from table...");
```

```
        cmd = new SqlCommand(@"SELECT * FROM [dbo]
          .[CustomTable]", connection);
        using (SqlDataReader reader = await
          cmd.ExecuteReaderAsync())
        {
          while (await reader.ReadAsync())
          {
            var id = reader.GetFieldValue<int>(0);
            var name = reader.GetFieldValue<string>(1);

            Console.WriteLine("Table row: Id {0}, Name {1}",
              id, name);
          }
        }
      }
    }
    catch(Exception ex)
    {
      Console.WriteLine("Error: {0}", ex.Message);
    }
  }
```

4. 在 Main 方法中加入以下代码片段：

```
const string dataBaseName = "CustomDatabase";
var t = ProcessAsynchronousIO(dataBaseName);
t.GetAwaiter().GetResult();
Console.WriteLine("Press Enter to exit");
Console.ReadLine();
```

5. 运行程序。

9.4.3 工作原理

该程序使用了一个软件，叫做 SQL Server 2012 LocalDb。安装 Visual Studio 2012 时会附带安装它，应该能正常使用。但是如果有什么错误，你可以通过安装向导来修复该组件。

先要配置数据库文件的存放路径。我们将数据库文件放置在应用程序执行目录中。有两个文件，一个是数据库本身，另一个是事务日志文件。我们也配置了两个连接字符串来定义如何连接数据库。第一个字符串是连接到 LocalDb 引擎来分离数据库。如果数据库已经存在，则删除并重建。当打开连接以及单独使用 OpenAsync 和 ExecuteNonQueryAsync 方法执行 SQL 命令时，我们使用了 I/O 异步操作。

在该任务完成后，我们附加了一个最新创建的数据库。我们创建了一张新的表并插入了一些数据。除了之前提到的方法，我们还使用了 ExecuteScalarAsync 来异步地从数据库引擎中得到一个标量值，并且使用 SqlDataReader.ReadAsync 方法来从数据库表中异步地读取数据行。

如果在数据库有一个大数据量的表，里面数据行中包含大数据量的二进制值，可以使用

CommandBehavior.SequentialAcess 枚举来创建数据阅读器，并使用 GetFieldValueAsync 方法异步地通过数据阅读器获取大字段值。

9.5　异步调用 WCF 服务

本节描述了如何创建一个 WCF 服务，并宿主在命令行应用程序中，客户端可以访问服务元数据，并以异步的方式消费它。

9.5.1　准备工作

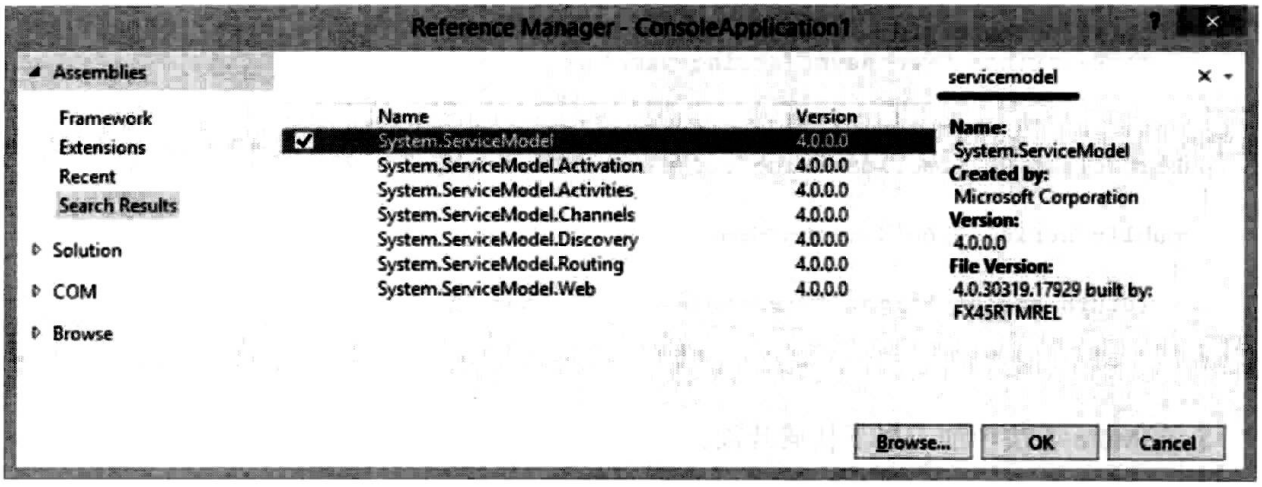
为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\Chapter9\Recipe4 目录中。

9.5.2　实现方式

请执行以下步骤来了解如何使用 WCF 服务：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 添加对 System.ServiceModel 库的引用。右键点击该项目的引用目录，选择**添加引用**菜单选项。添加对 System.ServiceModel 库的引用。你可以选择引用管理对话框中的搜索功能，如下面的截图所示：



3. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.ServiceModel;
using System.ServiceModel.Description;
using System.Threading.Tasks;
```

4. 在 Main 方法下面加入以下代码片段：

```
const string SERVICE_URL =
  "http://localhost:1234/HelloWorld";

static async Task RunServiceClient()
{
  var endpoint = new EndpointAddress(SERVICE_URL);
  var channel = ChannelFactory
    <IHelloWorldServiceClient>.CreateChannel(new
    BasicHttpBinding(), endpoint);

  var greeting = await channel.GreetAsync("Eugene");
  Console.WriteLine(greeting);
}

  [ServiceContract(Namespace = "Packt", Name =
    "HelloWorldServiceContract")]
public interface IHelloWorldService
{
  [OperationContract]
  string Greet(string name);
}

[ServiceContract(Namespace = "Packt", Name =
  "HelloWorldServiceContract")]
public interface IHelloWorldServiceClient
{
  [OperationContract]
    string Greet(string name);

  [OperationContract]
    Task<string> GreetAsync(string name);
}

public class HelloWorldService : IHelloWorldService
{
  public string Greet(string name)
  {
    return string.Format("Greetings, {0}!", name);
  }
}
```

5. 在 Main 方法中加入以下代码片段：

```
ServiceHost host = null;

try
{
  host = new ServiceHost(typeof (HelloWorldService), new
    Uri(SERVICE_URL));
  var metadata = host.Description.Behaviors.
    Find<ServiceMetadataBehavior>();
  if (null == metadata)
```

```
    {
      metadata = new ServiceMetadataBehavior();
    }

    metadata.HttpGetEnabled = true;
    metadata.MetadataExporter.PolicyVersion =
      PolicyVersion.Policy15;
    host.Description.Behaviors.Add(metadata);

    host.AddServiceEndpoint(ServiceMetadataBehavior
      .MexContractName, MetadataExchangeBindings
      .CreateMexHttpBinding(),"mex");
    var endpoint = host.AddServiceEndpoint(typeof
      (IHelloWorldService), new BasicHttpBinding(),
      SERVICE_URL);

    host.Faulted += (sender, e) =>
      Console.WriteLine("Error!");

    host.Open();

    Console.WriteLine("Greeting service is running and
      listening on:");
    Console.WriteLine("{0} ({1})", endpoint.Address,
      endpoint.Binding.Name);

    var client = RunServiceClient();
    client.GetAwaiter().GetResult();

    Console.WriteLine("Press Enter to exit");
    Console.ReadLine();
}
catch (Exception ex)
{

    Console.WriteLine("Error in catch block: {0}", ex);
}
finally
{
    if (null != host)
    {
      if (host.State == CommunicationState.Faulted)
      {
        host.Abort();
      }
      else
      {
        host.Close();
      }
    }
}
```

6. 运行程序。

9.5.3 工作原理

Windows Communication Foundation（简称 WCF）是一个框架，用于以不同的方式调用远程服务。其中一个有一段时间非常流行，用于通过 HTTP 使用基于 XML 的协议来调用远程服务，它叫做**简单对象访问协议**（Simple Object Access Protocol，简称 SOAP）。

Visual Studio 2012 对 WCF 服务有着非常丰富的支持。例如，你可以使用**添加服务引用**菜单项给这样的服务添加引用。你也可对本节中的服务使用此功能，因为我们提供了服务元数据。

为了创建这样的服务，我们需要使用 ServiceHost 类来宿主我们的服务。我们通过提供一个服务实现类型和服务地址 URL 来描述如何宿主服务。然后配置了元数据终端和服务终端。最后，使用 Faulted 事件来处理错误，并运行该宿主服务。

为了消费该服务，我们创建了一个客户端，这是主要的技巧所在。在服务器端，我们有一个服务，是一个普通的同步方法，叫做 Greet。服务契约 IHelloWorldService 定义了该方法。然而，如果想使用异步网络 I/O，我们需要异步地调用该方法。可以通过使用匹配的命名空间和服务名来创建一个新的服务契约，然后同时定义同步方法和基于任务的异步方法。尽管事实上在服务器端我们没有异步方法，但是如果我们遵循命名约定，WCF 基础设施明白我们想创建一个异步的代理方法。

因此，当我们创建一个 IHelloWorldServiceClient 代理渠道，WCF 会正确地路由一个异步调用到该服务器端同步方法。如果你运行程序，然后打开浏览器并使用该服务的 URL http://localhost:1234/HelloWorld 来访问该服务。你会看到该服务的描述，还可以浏览 XML 元数据，该元数据可用于从 Visual Studio 2012 添加服务引用。如果你尝试生成引用，将看到稍微有点复杂的代码，但它是自动创建的，并且易于使用。



第 10 章 Chapter 10

并行编程模式

在本章中，我们将综述程序员在实现并行工作流时经常会遇到的问题。你将学到以下内容：

- 实现惰性求值的共享状态
- 使用 BlockingCollection 实现并行管道
- 使用 TPL 数据流实现并行管道
- 使用 PLINQ 实现 Map/Reduce 模式

10.1 简介

编程中的模式是指针对既定问题的具体的标准的方案。通常，编程模式是人们总结经验的结果，即分析常见的问题，并对这些问题提供方案。

由于并行编程已经存在了很长一段时间，有很多不同的模式来编写并行的应用程序。甚至有特殊的编程语言更加容易地编写特定并行算法。然而，这正是事情变得日益复杂的源头。在本书中，我将提供一个起点从而帮助你能够进一步学习并行编程。我们将回顾非常基础的但是非常有用的模式，这对于解决异步编程的很多常见的方案很有帮助。

首先是关于在多线程中使用**共享状态的对象**。我想强调你应当尽可能地避免使用它。我们在之前章节中已经讨论过了，当编写并行算法时共享状态是非常不好的方式，但在很多场合中它又是不可避免的。我们将学习如何延迟一个对象的计算直到需要时才进行计算，以及如何在不同的场景中实现线程安全。

接下来的两节将展示如何创建一个结构化的并行数据流。我们将回顾生产者 / 消费者模

式的一个具体实例，其被称为**并行管道**（Parallel Pipeline）。我们先通过阻塞集合来实现它，然后看看微软提供的另一个并行编程库——**TPL 数据流**（TPL DataFlow）是多么有用。

我们将学习的最后一个模式是 Map/Reduce 模式。现代世界中，该名字可以指很多事情。一些人认为 map/reduce 不是针对任何问题的通用方式，而是大型的分布式的集群计算的具体实现。我们将找出该名称背后的含义，以及学习如何在小型的并行应用程序中使用该模式。

10.2 实现惰性求值的共享状态

本节展示了如何编写一个具有惰性求值特点的线程安全的共享状态对象。

10.2.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 BookSamples\ Chapter10\Recipe1 目录中。

10.2.2 实现方式

请执行以下步骤来实现一个惰性求值的共享状态状态：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
static async Task ProcessAsynchronously()
{
  var unsafeState = new UnsafeState();
  Task[] tasks = new Task[4];

  for (int i = 0; i < 4; i++)
  {
    tasks[i] = Task.Run(() => Worker(unsafeState));
  }
  await Task.WhenAll(tasks);
  Console.WriteLine(" -------------------------- ");

  var firstState = new DoubleCheckedLocking();
  for (int i = 0; i < 4; i++)
  {
    tasks[i] = Task.Run(() => Worker(firstState));
  }

  await Task.WhenAll(tasks);
```

```
  Console.WriteLine(" -------------------------- ");

  var secondState = new BCLDoubleChecked();
  for (int i = 0; i < 4; i++)
  {
    tasks[i] = Task.Run(() => Worker(secondState));
  }

  await Task.WhenAll(tasks);
  Console.WriteLine(" -------------------------- ");

  var thirdState = new Lazy<ValueToAccess>(Compute);
  for (int i = 0; i < 4; i++)
  {
    tasks[i] = Task.Run(() => Worker(thirdState));
  }

  await Task.WhenAll(tasks);
  Console.WriteLine(" -------------------------- ");

  var fourthState = new BCLThreadSafeFactory();
  for (int i = 0; i < 4; i++)
  {
    tasks[i] = Task.Run(() => Worker(fourthState));
  }

  await Task.WhenAll(tasks);
  Console.WriteLine(" -------------------------- ");

}

static void Worker(IHasValue state)
{
  Console.WriteLine("Worker runs on thread id {0}",
    Thread.CurrentThread.ManagedThreadId);
  Console.WriteLine("State value: {0}", state.Value.Text);
}

static void Worker(Lazy<ValueToAccess> state)
{
  Console.WriteLine("Worker runs on thread id {0}",
    Thread.CurrentThread.ManagedThreadId);
  Console.WriteLine("State value: {0}", state.Value.Text);
}

static ValueToAccess Compute()
{
  Console.WriteLine("The value is being constructed on a
    thread id {0}", Thread.CurrentThread.ManagedThreadId);
  Thread.Sleep(TimeSpan.FromSeconds(1));
  return new ValueToAccess(string.Format(
    "Constructed on thread id {0}",
      Thread.CurrentThread.ManagedThreadId));
}

class ValueToAccess
{
```

```
  private readonly string _text;
  public ValueToAccess(string text)
  {
    _text = text;
  }

  public string Text
  {
    get { return _text; }
  }
}

class UnsafeState : IHasValue
{
  private ValueToAccess _value;

  public ValueToAccess Value
  {
    get
    {
      if (_value == null)
      {
        _value = Compute();
      }
      return _value;
    }
  }


}

class DoubleCheckedLocking : IHasValue
{
  private object _syncRoot = new object();
  private volatile ValueToAccess _value;

  public ValueToAccess Value
  {
    get
    {
      if (_value == null)
      {
        lock (_syncRoot)
        {
          if (_value == null) _value = Compute();
        }
      }
      return _value;
    }
  }
}

class BCLDoubleChecked : IHasValue
{
  private object _syncRoot = new object();
  private ValueToAccess _value;
  private bool _initialized = false;
```

```
  public ValueToAccess Value
  {
    get
    {
      return LazyInitializer.EnsureInitialized(
        ref _value, ref _initialized, ref _syncRoot,
          Compute);
    }
  }
}

class BCLThreadSafeFactory : IHasValue
{
  private ValueToAccess _value;

  public ValueToAccess Value
  {
    get
    {
      return LazyInitializer.EnsureInitialized(ref _value,
        Compute);
    }
  }
}

interface IHasValue
{
  ValueToAccess Value { get; }
}
```

4. 在 Main 方法中加入以下代码片段：

```
var t = ProcessAsynchronously();
t.GetAwaiter().GetResult();

Console.WriteLine("Press ENTER to exit");
Console.ReadLine();
```

5. 运行程序。

10.2.3　工作原理

第一个例子展示了在多个访问线程中为什么使用 UnsafeState 对象是不安全的。可以看到构造方法被调用了多次，并且不同的线程中值是不同的，显而易见这是错误的。为了修复该问题，我们可以在读取值时使用锁，如果值还未初始化，则先创建。这虽然可行，但是对每个读操作都使用锁非常低效。为了避免每次都使用锁，有一个传统的方式叫做**双重锁定**（double-checked locking）模式。第一次检查该值时，如果它不是 null，我们避免了没有必要的锁来使用共享对象。然而，如果它还未被构造，我们使用锁并再一次检查该值，因为它可能在第一次检查和锁定操作之间被初始化。如果仍未被初始化，那么我们计算该值。我们可以看到第二个示例使用了该方式。对构造方法只有一次调用，并且第一个调用的线程定义了该共享对象状态。



请记住如果惰性求值的对象实现是线程安全的，并不自动说明它的所有属性也是线程安全的。

例如，如果你给 ValueToAccess 对象添加一个 int 公共属性，它则不是线程安全的。你仍需要使用互锁结构（interlocked construct）或锁来保证线程安全。

该模式很常见，这也是为什么在基础类库中有一些类来方便我们使用。首先，我们可以使用 LazyInitializer.EnsureInitialized 方法，其内部实现了双重锁定模式。然而，最舒服的选项是使用 Lazy<T> 类。它可以让我们立即拥有可用的线程安全并且惰性求值的共享状态对象。接下来的两个例子展示了它们与第二个例子几乎相同，程序行为也一致。唯一不同是由于 LazyInitializer 是一个静态类，因此我们无需创建一个类的实例，而 Lazy<T> 中例子则需要，因此前者的性能在某些场景中更好。

最后一个选择是如果我们不关心构造方法，则完全可以避免使用锁。如果对象是线程安全的并且没有副作用和严重的性能影响，我们可以只使用第一个构造值运行多次。最后一个例子展示了该行为，我们可以使用另一个 LazyInitializer. EnsureInitialized 方法重载达到目的。

10.3 使用 BlockingCollection 实现并行管道

本节将描述如何使用标准的 BlockingCollection 数据结构实现一个生产者 / 消费者模式的特定场景。该特定场景被称为并行管道。

10.3.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 7644_Code\Chapter10\Recipe2 目录中。

10.3.2 实现方式

请执行以下步骤来了解如何使用 BlockingCollection 来实现并行管道：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Collections.Concurrent;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
```

3. 在 Main 方法下面加入以下代码片段：

```
private const int CollectionsNumber = 4;
private const int Count = 10;

class PipelineWorker<TInput, TOutput>
{
  Func<TInput, TOutput> _processor = null;
  Action<TInput> _outputProcessor = null;
  BlockingCollection<TInput>[] _input;
  CancellationToken _token;

  public PipelineWorker(
      BlockingCollection<TInput>[] input,
      Func<TInput, TOutput> processor,
      CancellationToken token,
      string name)
  {
    _input = input;
    Output = new BlockingCollection<TOutput>[_input.Length];
    for (int i = 0; i < Output.Length; i++)
      Output[i] = null == input[i] ? null : new BlockingCollection
<TOutput>(Count);

    _processor = processor;
    _token = token;
    Name = name;
  }

  public PipelineWorker(
      BlockingCollection<TInput>[] input,
      Action<TInput> renderer,
      CancellationToken token,
      string name)
  {
    _input = input;
    _outputProcessor = renderer;
    _token = token;
    Name = name;
    Output = null;
  }

  public BlockingCollection<TOutput>[] Output { get; private set;
}

  public string Name { get; private set; }

  public void Run()
  {
    Console.WriteLine("{0} is running", this.Name);
    while (!_input.All(bc => bc.IsCompleted) && !_token.
IsCancellationRequested)
    {
```

```
        TInput receivedItem;
        int i = BlockingCollection<TInput>.TryTakeFromAny(
            _input, out receivedItem, 50, _token);
        if (i >= 0)
        {
          if (Output != null)
          {
            TOutput outputItem = _processor(receivedItem);
            BlockingCollection<TOutput>.AddToAny(Output,
              outputItem);
            Console.WriteLine("{0} sent {1} to next,
              on thread id {2}", Name, outputItem,
                Thread.CurrentThread.ManagedThreadId);
            Thread.Sleep(TimeSpan.FromMilliseconds(100));
          }
          else
          {
            _outputProcessor(receivedItem);
          }
        }
        else
        {
          Thread.Sleep(TimeSpan.FromMilliseconds(50));
        }
      }
      if (Output != null)
      {
        foreach (var bc in Output) bc.CompleteAdding();
      }
    }
}
```

4. 在 Main 方法中加入以下代码片段：

```
var cts = new CancellationTokenSource();

Task.Run(() =>
{
  if (Console.ReadKey().KeyChar == 'c')
    cts.Cancel();
});

var sourceArrays = new BlockingCollection<int>[
  CollectionsNumber];
for (int i = 0; i < sourceArrays.Length; i++)
{
  sourceArrays[i] = new BlockingCollection<int>(Count);
}

var filter1 = new PipelineWorker<int, decimal>
(sourceArrays,
  (n) => Convert.ToDecimal(n * 0.97),
```

```
  cts.Token,
  "filter1"
);

var filter2 = new PipelineWorker<decimal, string>
(filter1.Output,
  (s) => String.Format("--{0}--", s),
  cts.Token,
  "filter2"
  );

var filter3 = new PipelineWorker<string, string>
(filter2.Output,
  (s) => Console.WriteLine("The final result is {0} on
    thread id {1}", s,
      Thread.CurrentThread.ManagedThreadId), cts.Token,
        "filter3");

try
{
  Parallel.Invoke(
    () =>
    {
      Parallel.For(0, sourceArrays.Length * Count,
        (j, state) =>
      {
        if (cts.Token.IsCancellationRequested)
        {
          state.Stop();
        }
        int k = BlockingCollection<int>.TryAddToAny(
          sourceArrays, j);
        if (k >= 0)
        {
          Console.WriteLine("added {0} to source data on
            thread id {1}", j,
              Thread.CurrentThread.ManagedThreadId);
          Thread.Sleep(TimeSpan.FromMilliseconds(100));
        }
      });
      foreach (var arr in sourceArrays)
      {
        arr.CompleteAdding();
      }
    },
    () => filter1.Run(),
    () => filter2.Run(),
    () => filter3.Run()
  );
}
catch (AggregateException ae)
{
  foreach (var ex in ae.InnerExceptions)
    Console.WriteLine(ex.Message + ex.StackTrace);
```

```
}

if (cts.Token.IsCancellationRequested)
{
  Console.WriteLine("Operation has been canceled!
    Press ENTER to exit.");
}
else
{
  Console.WriteLine("Press ENTER to exit.");
}
Console.ReadLine();
```

5. 运行程序。

10.3.3 工作原理

前面的示例中，我们实现了最常见的异步编程场景的其中之一。假设我们有一些数据需要经历数个计算阶段，并且这些计算要花费大量时间。后者计算需要使用前者的结果，所以不能并行运行它们。

如果只有一个项要处理，则可能很难改进性能。然而，如果许多项要经历相同的计算阶段，我们可以使用并行管道技术。这意味着我们不用等待所有项通过第一个计算阶段，然后才进行第二个。其实只需一个项完成计算阶段后，就可以将其移动到下一个阶段，同时接下来的项可以被之前的阶段处理，依次类推。结果，通过移动第一个项在第一个计算阶段的时间来实现一个几乎是并行的处理过程。

这里，我们为每个处理阶段使用了 4 个集合，这说明了我们也可以以并行的方式处理每个阶段。我们做的第一步是提供了按下 C 键来取消整个过程的功能。我们创建了一个取消标志，并运行一个单独的任务来监视 C 键。然后我们自定义了管道。它由三个主要的阶段组成。第一个阶段是将初始值放入到前四个集合中，其将被作为源数据被下一个阶段使用。该代码在 Parallel.For 循环中，而该循环在 Parallel.Invoke 声明中。由于我们以并行的方式运行所有阶段，所以这初始化阶段也是以并行的方式运行。

接下来的阶段是定义管道元素。该逻辑定义在 PipelineWorker 类中。我们使用输入集合初始化了工作者，提供了一个转换函数，然后和其他的工作者一起并行运行。这里我们定义了两个工作者，也可以称为过滤器，因为它们过滤了初始序列。其中一个将整数转换为小数，第二个将小数转换为字符串。最后一个工作者只将每个传入的字符串打印到控制台。每个地方都提供了运行的线程 ID，来揭示整个程序是如何工作的。此外，我们添加了人为的延迟，所以每项的处理更加自然，就像我们真的需要大计算量一样。

结果我们得到了实际的期望行为。首先，一些项被创建在初始集合中。接下来，第一个过滤器可以处理它们，待处理完毕后，第二个处理器也开始工作，最终该项被传给最后一个工作者并将其打印到控制台。

10.4　使用 TPL 数据流实现并行管道

本节展示了如何借助于 TPL 数据流库来实现一个并行管道模式。

10.4.1　准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 7644_Code\Chapter10\Recipe3 目录中。

10.4.2　实现方式

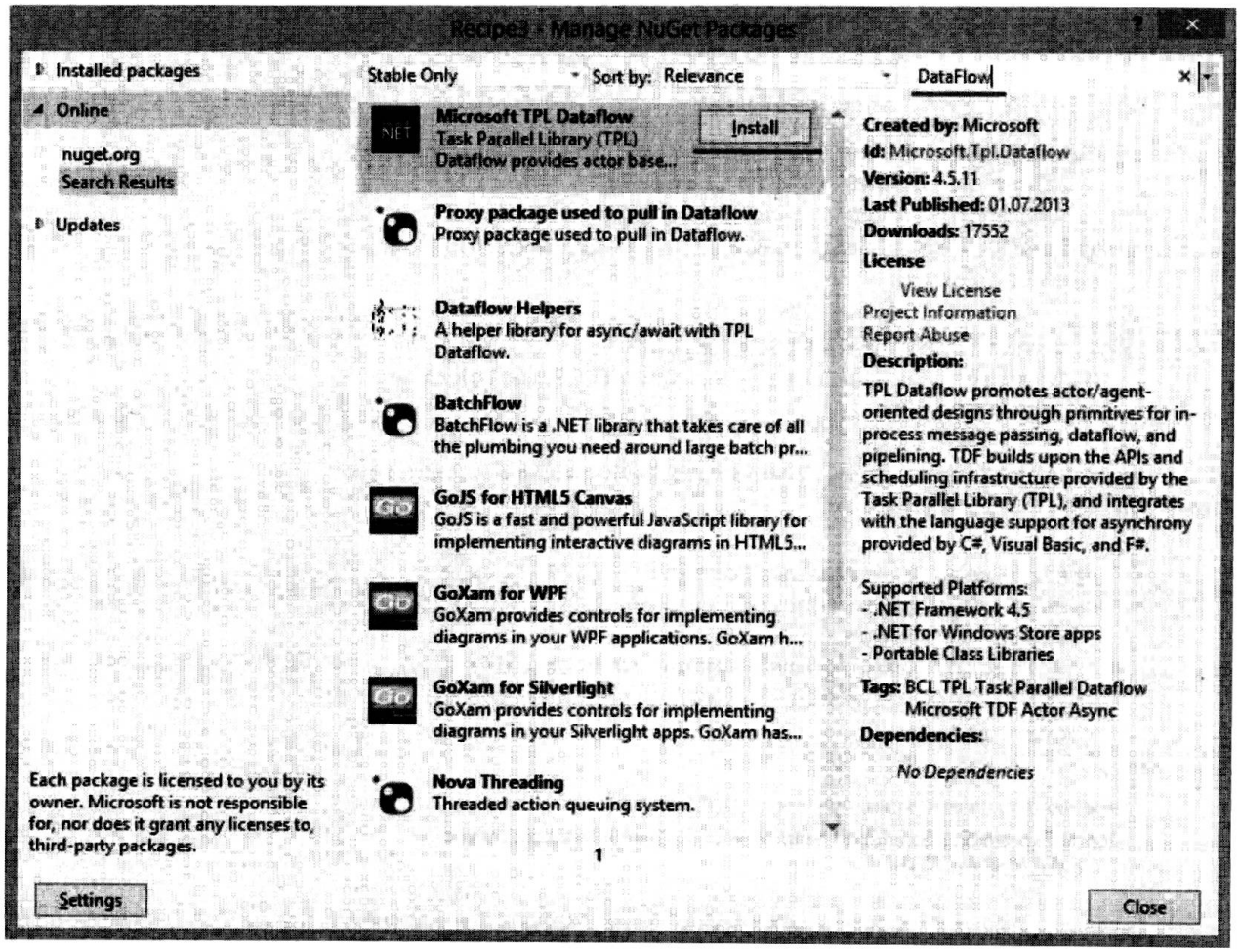
请执行以下步骤来了解如何使用 TPL 数据流实现并行管道：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 添加对 Microsoft TPL DataFlow NuGet 包的引用。

1）右键点击项目的**引用**目录，并选择**管理 NuGet 包**…菜单项。

2）添加对 Microsoft TPL DataFlow NuGet 包的引用。你可以使用**管理 NuGet 包**对话框的搜索选项，如下图所示：



3. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
```

4. 在 Main 方法下面加入以下代码片段：

```
async static Task ProcessAsynchronously()
{
  var cts = new CancellationTokenSource();

  Task.Run(() =>
  {
    if (Console.ReadKey().KeyChar == 'c')
      cts.Cancel();
  });

  var inputBlock = new BufferBlock<int>(
    new DataflowBlockOptions { BoundedCapacity = 5,
      CancellationToken = cts.Token });

  var filter1Block = new TransformBlock<int, decimal>(
    n =>
    {
      decimal result = Convert.ToDecimal(n * 0.97);
      Console.WriteLine("Filter 1 sent {0} to the next
        stage on thread id {1}", result,
          Thread.CurrentThread.ManagedThreadId);
      Thread.Sleep(TimeSpan.FromMilliseconds(100));
      return result;
    },
    new ExecutionDataflowBlockOptions {
      MaxDegreeOfParallelism = 4, CancellationToken =
        cts.Token });

  var filter2Block = new TransformBlock<decimal, string>(
    n =>
    {
      string result = string.Format("--{0}--", n);
      Console.WriteLine("Filter 2 sent {0} to the next
        stage on thread id {1}", result,
          Thread.CurrentThread.ManagedThreadId);
      Thread.Sleep(TimeSpan.FromMilliseconds(100));
      return result;
    },
    new ExecutionDataflowBlockOptions {
     MaxDegreeOfParallelism = 4, CancellationToken =
       cts.Token });

  var outputBlock = new ActionBlock<string>(
```

```
    s =>
    {
      Console.WriteLine("The final result is {0} on thread
        id {1}", s, Thread.CurrentThread.ManagedThreadId);
    },
    new ExecutionDataflowBlockOptions {
      MaxDegreeOfParallelism = 4, CancellationToken =
        cts.Token });

  inputBlock.LinkTo(filter1Block, new DataflowLinkOptions {
    PropagateCompletion = true });
  filter1Block.LinkTo(filter2Block, new DataflowLinkOptions
    { PropagateCompletion = true });
  filter2Block.LinkTo(outputBlock, new DataflowLinkOptions
    { PropagateCompletion = true });

  try
  {
    Parallel.For(0, 20, new ParallelOptions {
      MaxDegreeOfParallelism = 4, CancellationToken =
        cts.Token }
    , i =>
    {
      Console.WriteLine("added {0} to source data on thread
        id {1}", i, Thread.CurrentThread.ManagedThreadId);
      inputBlock.SendAsync(i).GetAwaiter().GetResult();
    });
    inputBlock.Complete();
    await outputBlock.Completion;
    Console.WriteLine("Press ENTER to exit.");
  }
  catch (OperationCanceledException)
  {
    Console.WriteLine("Operation has been canceled!
      Press ENTER to exit.");
  }

  Console.ReadLine();
}
```

5. 在 Main 方法中加入以下代码片段：

```
var t = ProcessAsynchronously();
t.GetAwaiter().GetResult();
```

6. 运行程序。

10.4.3　工作原理

在之前的小节中，我们实现了一个并行管道模式以顺序阶段的方式处理元素。这是个很常见的问题，编程实现该算法的其中一个被推荐的方式是使用微软提供的 TPL 数据流库。该

库通过 NuGet 分发，所以很容易在应用程序中安装和使用。

TPL 数据流库包含不同类型的块，这些块可以根据复杂的处理过程以不同的方式连接，这些过程可以按需并行运行或顺序运行。为了展示一些可用的基础设施，我们借助于 TPL 数据流库实现了前一小节中的场景。

首先我们定义了不同的块来处理数据。请注意这些块有不同的选项，可以通过各自的构造函数来指定。这些选项是非常重要的。例如，我们给每个定义的块中传入取消标志，当发出取消信号时，所有块将停止工作。

先从 BufferBlock 开始处理。这块储存元素将元素传给流中的下一个块。我们通过指定 BoundedCapacity 选项值，强制其容量最多为 5 个元素。这意味着当已经有 5 个元素在该块中时，它将停止接收新元素，直到一个现有元素被传送给下一个块。

接下来的块类型是 TransformBlock。该块用于数据转换步骤。我们定义了两个转换块，其中一个将整数转换为小数，第二个将小数值转换为字符串。该块有一个 MaxDegree-OfParallelism 选项，指定了同时运行的最大工作线程数。

最后一个块是 ActionBlock 类型。该块将对每个传入的元素运行一个指定的操作。

现在，我们使用 LinkTo 方法把这些块连接到一起。这里我们拥有的是一个简单的顺序的数据流，但创建一个更复杂的数据流的模板也是可行的。这里我们通过给 PropagateCompletion 设置为 true 提供了一个 DataflowLinkOptions 选项。这意味着当步骤完成时，将自动将结果和异常传播到下一个阶段。我们开始并行地向缓冲块中添加项，并且当完成添加元素后调用块的 Complete 方法。然后等待最后的块来完成。一旦有取消标志发出，我们可以处理 OperationCancelledException，并且取消整个进程。

10.5 使用 PLINQ 实现 Map/Reduce 模式

本节将描述如何使用 PLINQ 来实现 Map/Reduce 模式。

10.5.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012。除此之外无需其他准备。本节的源代码放置在 7644_Code\Chapter10\Recipe4 目录中。

10.5.2 实现方式

请执行以下步骤来学习如何使用 PLINQ 实现 Map/Reduce 模式：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
```

3. 在 Main 方法下面加入以下代码片段：

```
private static readonly char[] delimiters =
  Enumerable.Range(0, 256).Select(i => (char)i).Where(c =>
    !char.IsLetterOrDigit(c)).ToArray();

private const string textToParse = @"
Call me Ishmael. Some years ago - never mind how long precisely -
having little or no money in my purse, and nothing particular to
interest me on shore, I thought I would sail about a little and
see the watery part of the world. It is a way I have of driving
off the spleen, and regulating the circulation. Whenever I find
myself growing grim about the mouth; whenever it is a damp,
drizzly November in my soul; whenever I find myself involuntarily
pausing before coffin warehouses, and bringing up the rear of
every funeral I meet; and especially whenever my hypos get such
an upper hand of me, that it requires a strong moral principle
to prevent me from deliberately stepping into the street, and
methodically knocking people's hats off - then, I account it high
time to get to sea as soon as I can.

☒ Herman Melville, Moby Dick.
";
```

4. 在 Main 方法中加入以下代码片段：

```
var q = textToParse.Split(delimiters)
  .AsParallel()
  .MapReduce(
    s => s.ToLower().ToCharArray()
  , c => c
  , g => new[] {new {Char = g.Key, Count = g.Count()}})
  .Where(c => char.IsLetterOrDigit(c.Char))
  .OrderByDescending( c => c.Count);

foreach (var info in q)
{
  Console.WriteLine("Character {0} occured in the text {1}
    {2}", info.Char, info.Count, info.Count == 1 ? "
      time" : "times");
}
Console.WriteLine(" ----------------------------------------");
const string searchPattern = "en";

var q2 = textToParse.Split(delimiters)
  .AsParallel()
  .Where(s => s.Contains(searchPattern))
```

```
  .MapReduce(
    s => new [] {s}
    , s => s
    , g => new[] {new {Word = g.Key, Count = g.Count()}})
  .OrderByDescending(s => s.Count);

Console.WriteLine("Words with search pattern '{0}':",
  searchPattern);
foreach (var info in q2)
{
  Console.WriteLine("{0} occured in the text {1} {2}",
    info.Word, info.Count,
    info.Count == 1 ? "time" : "times");
}

int halfLengthWordIndex = textToParse.IndexOf(' ',
  textToParse.Length/2);

using(var sw = File.CreateText("1.txt"))
{
  sw.Write(textToParse.Substring(0, halfLengthWordIndex));
}

using(var sw = File.CreateText("2.txt"))
{
  sw.Write(textToParse.Substring(halfLengthWordIndex));
}

string[] paths = new[] { ".\\" };

Console.WriteLine(" ----------------------------------------------
--");
var q3 = paths
  .SelectMany(p => Directory.EnumerateFiles(p, "*.txt"))
  .AsParallel()
  .MapReduce(
    path => File.ReadLines(path).SelectMany(line =>
      line.Trim(delimiters).Split(delimiters)),
        word => string.IsNullOrWhiteSpace(word) ? '\t' :
          word.ToLower()[0], g => new [] { new {
            FirstLetter = g.Key, Count = g.Count()}})
  .Where(s => char.IsLetterOrDigit(s.FirstLetter))
  .OrderByDescending(s => s.Count);

Console.WriteLine("Words from text files");

foreach (var info in q3)
{
  Console.WriteLine("Words starting with letter '{0}'
    occured in the text {1} {2}", info.FirstLetter,
      info.Count,
    info.Count == 1 ? "time" : "times");
}
```

5. 在 Program 类定义后添加下列代码片段：

```
static class PLINQExtensions
{
  public static ParallelQuery<TResult> MapReduce<TSource,
    TMapped, TKey, TResult>(
    this ParallelQuery<TSource> source,
    Func<TSource, IEnumerable<TMapped>> map,
    Func<TMapped, TKey> keySelector,
    Func<IGrouping<TKey, TMapped>,
    IEnumerable<TResult>> reduce)
  {
    return source.SelectMany(map)
    .GroupBy(keySelector)
    .SelectMany(reduce);
  }
}
```

6. 运行程序。

10.5.3　工作原理

Map/Reduce 功能是另一个重要的并行编程模式。它适用于小程序，及拥有大量的多个服务器端计算的场景。该模式的含义是你有两个特殊的功能要应用于你的数据。其中第一个是 Map 函数。它接收一组键 / 值列表的初始数据，并产生另一组键 / 值序列，将初始数据转换为适合的格式以便进行下一步处理。然后我们使用另一个叫做 Reduce 的函数。Reduce 函数使用 Map 函数的结果，并将其转换为我们真正需要的尽可能小的数据集。为了了解该算法如何工作，让我们看看本节的示例。

首先我们在字符串变量 textToParse 中定义了一段相当大的文本。我们需要该段文本来运行查询。然后定义了 Map/Reduce 的实现，将其作为 PLINQExtensions 类中的一个 PLINQ 扩展方法。我们使用 SelectMany 来将初始序列转换为需要应用于 Map 函数的序列。该函数从一个序列元素中产生多个新的元素。然后我们选择如何使用 keySelector 函数来对新序列分组，并且使用该键以及 GroupBy 来产生一个中间键 / 值序列。最后一件事是对分组后的序列应用 Reduce 函数来得到结果。

在第一个例子中，我们将该文本分割成单词，然后借助于 Map 函数将每个单词拆分为单个字符，最后根据字符值进行分组。Reduce 函数最终将序列转换为键值对，其中键是字符，值为该字符在该段文本中被使用过的次数，并且该序列按照被使用次数进行排序。因此，我们能够并行地计算该文本中每个字符出现的次数（这是因为我们使用 PLINQ 来查询初始数据）。

接下来的例子非常简单，我们使用 PLINQ 来过滤并保留只包含搜索模式的单词，然后再计算这些单词在该文本中的使用次数。

最后一个例子使用了文件 I/O。我们将示例文本保存到磁盘，将其分割为两个文件。然后我们定义 Map 函数来根据目录名产生一组字符串，目录名是初始目录中所有文本文件中所有行中的所有单词。然后根据单词第一个字母对这些单词进行分组（过滤掉空字符串），然后使用 reduce 来看文本中那个字母作为单词首字母使用次数最多。这样的好处在于我们可以使用 map 和 reduce 函数的其他实现来轻易修改和分发该程序。并且仍然可以使用 PLINQ 来实现 map 和 reduce 函数，保证程序容易阅读和维护。



第 11 章 Chapter 11

更多信息

在本章中，我们将学习一些在 Windows 8 操作系统中使用的一些新的编程范例。你将学习一下内容：

- 在 Windows 商店应用中使用计时器
- 在通常的应用程序中使用 WinRT
- 在 Windows 商店应用中使用 BackgroundTask

11.1 简介

微软在 2011 年 9 月 13 日举行的 BUILD 大会上发布了 Windows 8 的第一个公共测试版本。该新的操作系统试图解决 Windows 的几乎所有问题。其引入的功能包括适用于可触摸的平板设备的响应式 UI，低耗电量，新的程序模型，新的异步 API，以及周密的安全措施。

Windows API 的核心改进是一个新的多平台组件系统，称为 WinRT。WinRT 是 COM 的逻辑演进。程序员借助于 WinRT 可以使用原生的 C++ 代码、C# 和 .NET，甚至 JavaScript 和 HTML 来开发应用程序。另一个改变是引入了中央应用程序商店，之前的 Windows 平台并不具有此功能。

Windows 8 作为新的应用程序平台，是向后兼容的，并且能够运行通常的 Windows 应用程序。这就导致了这样的情况，系统中主要存在两类应用程序：Windows 商店应用，其通过

Windows 商店进行分发，以及通常的经典的应用程序，这是由以前版本的 Windows 遗留下来的程序。

本章中我们主要关注 Windows 商店应用。这种应用开发模式改变了很多，但是作为程序员必须遵守特定的规则。该应用需要在有限的时间内启动或完成，保证整个操作系统和其他应用始终能够响应。为了节省电池电量，应用程序不再默认后台运行，则是会被挂起并实际停止执行。

新的 Windows API 是异步的，你只能在应用程序中使用白名单中的 API 函数。例如，不再允许你创建一个新的线程类。你只能使用系统管理的线程池。很多有用的 API 不能被使用，你不得不学习新的方式来达到与以前一样的目的。

在本章中，我们将了解 Windows 商店应用与通常的 Windows 应用程序有何不同，我们如何使用通常的应用程序中结合 WinRT 的优势，以及学习一个使用了后台通知的 Windows 商店应用的简化场景。

11.2 在 Windows 商店应用中使用计时器

本节展示了如何在 Windows 商店应用中使用简单的计时器。

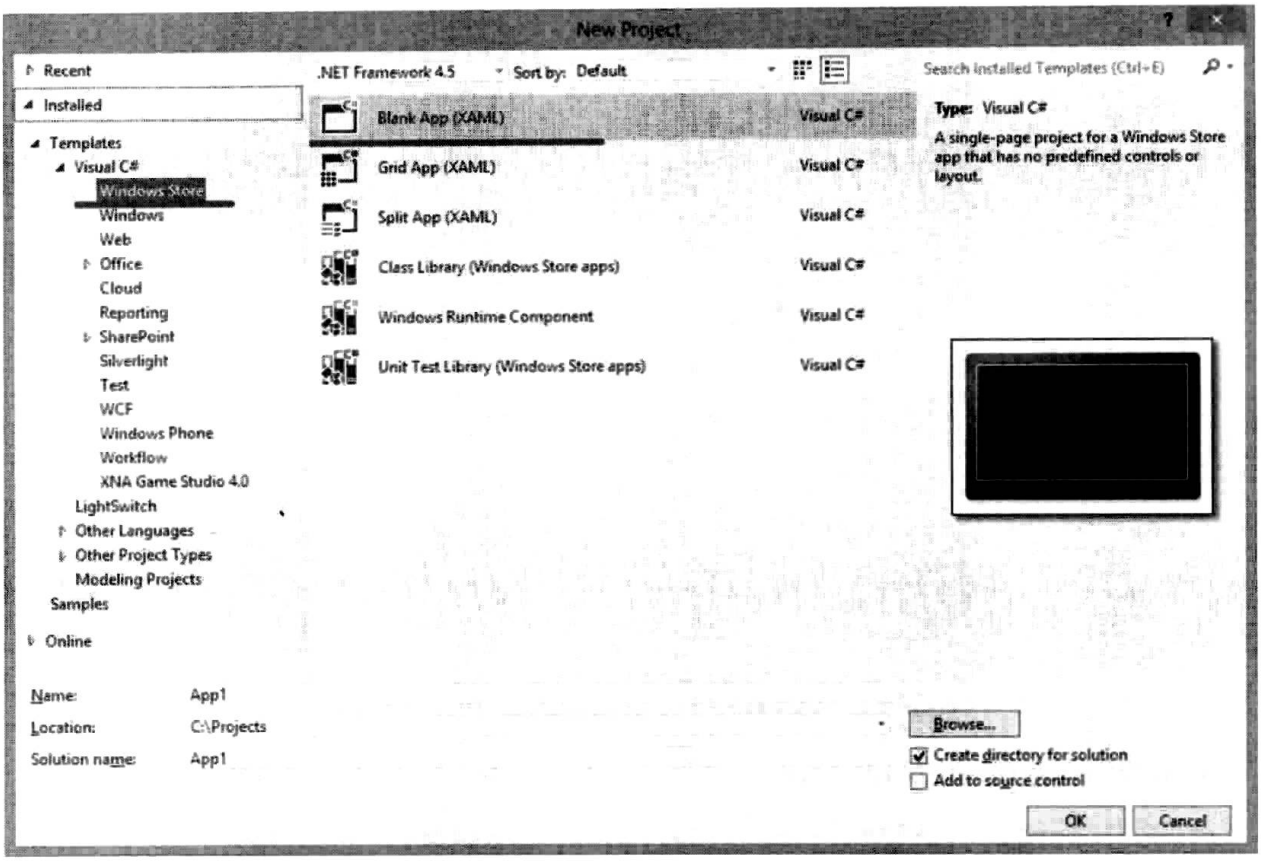
11.2.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012 和 Windows8（或以上）操作系统。除此之外无需其他准备。本节的源代码放置在 7644_Code\Chapter11\Recipe1 目录中。

11.2.2 实现方式

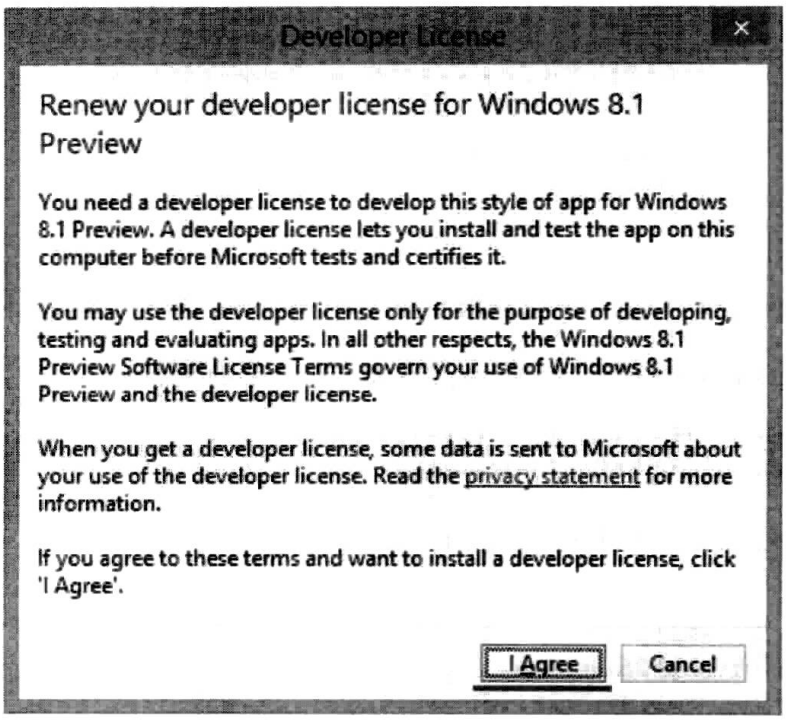
请执行以下步骤来了解如何在 Windows 商店应用中使用计时器。

1. 启动 Visual Studio 2012。新建一个 Windows Store 类别下的 C# **空白应用**（XAML）项目。



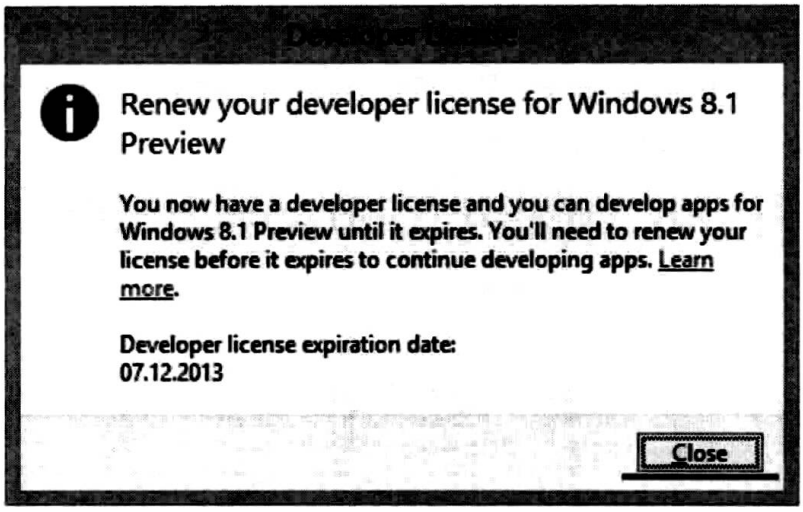
2. 如果询问你是否更新你的开发人员许可证，你必须同意该微软隐私条款。

3. 登录你的微软账号（或先创建一个）。





4. 最后你会得到一个确定对话框，说明开发人员许可证已经被成功更新。



5. 在 MainPage.xaml 文件中，给 Grid 元素加入一下名称属性：

```
<Grid Name="Grid" Background="{StaticResource
  ApplicationPageBackgroundThemeBrush}">
```

6. 在 MainPage.xaml.cs 文件中加入以下 using 指令：

```
using System;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;
```

7. 在 MainPage 构造函数定义下方加入以下代码片段：

```
private readonly DispatcherTimer _timer;
private int _ticks;
```

8. 使用下列代码片段替换掉 MainPage() 构造函数：

```
public MainPage()
{
  InitializeComponent();
  _timer = new DispatcherTimer();
  _ticks = 0;
}
```

9. 在 OnNavigatedTo 方法中加入以下代码片段：

```
Grid.Children.Clear();
var commonPanel = new StackPanel
{
  Orientation = Orientation.Vertical,
  HorizontalAlignment = HorizontalAlignment.Center
};

var buttonPanel = new StackPanel
{
  Orientation = Orientation.Horizontal,
  HorizontalAlignment = HorizontalAlignment.Center
};

var textBlock = new TextBlock
{
  Text = "Sample timer application",
  FontSize = 32,
  HorizontalAlignment = HorizontalAlignment.Center,
  Margin = new Thickness(40)
};

var timerTextBlock = new TextBlock
{
  Text = "0",
  FontSize = 32,
  HorizontalAlignment = HorizontalAlignment.Center,
  Margin = new Thickness(40)
};

var timerStateTextBlock = new TextBlock
{
  Text = "Timer is enabled",
  FontSize = 32,
  HorizontalAlignment = HorizontalAlignment.Center,
  Margin = new Thickness(40)
};

var startButton = new Button { Content = "Start",
  FontSize = 32};
```

```
var stopButton = new Button { Content = "Stop",
  FontSize = 32};

buttonPanel.Children.Add(startButton);
buttonPanel.Children.Add(stopButton);
commonPanel.Children.Add(textBlock);
commonPanel.Children.Add(timerTextBlock);
commonPanel.Children.Add(timerStateTextBlock);
commonPanel.Children.Add(buttonPanel);

_timer.Interval = TimeSpan.FromSeconds(1);
_timer.Tick += (sender, eventArgs) =>
{
  timerTextBlock.Text = _ticks.ToString(); _ticks++;
};
_timer.Start();

startButton.Click += (sender, eventArgs) =>
{
  timerTextBlock.Text = "0";
  _timer.Start();
  _ticks = 1;
  timerStateTextBlock.Text = "Timer is enabled";
};

stopButton.Click += (sender, eventArgs) =>
{
  _timer.Stop();
  timerStateTextBlock.Text = "Timer is disabled";
};

Grid.Children.Add(commonPanel);
```

10. 运行程序。

11.2.3 工作原理

当程序运行时创建了 MainPage 类的一个实例。在构造函数中实例化了 DispatcherTimer，并且初始化计时器值为 0。然后在 OnNavigatedTo 事件处理器中，创建了 UI 控制器并给开始和停止按钮绑定了相应的 lambda 表达式，表达式包含了开始和停止逻辑。

正如你看到的一样，计时器事件处理器直接作用于 UI 控制器。这是可行的，因为 DispatcherTimer 的实现方式是计时器的标记事件运行于 UI 线程中。然而，如果你运行该程序并切换到其他地方，几分钟后再切换回该程序，你会发现秒数远远小于经过的真实时间。发生这种情况是因为 Windows 8 应用程序（或通常称为 Windows 8 商店应用）有着完全不同的生命周期。



请注意 Windows 商店应用行为表现必须与智能手机或平板电脑上的应用一样。它们过段时间后不是在后台运行，而是会被挂起，这意味着除非用户切换回该应用，否则会一直保持冻结状态。在应用被挂起前你有有限的时间来保存当前程序状态，从而在该应用重新运行时你能够恢复到之前的状态。

该行为会节省电量和 CPU 资源，但在实现一些需要在后台运行的程序时会非常困难。Windows 8 有一组特殊的 API 来实现这种应用。稍后在本章中我们会学习一个这样的场景。

11.3　在通常的应用程序中使用 WinRT

该节展示了如何创建一个使用了 WinRT API 的命令行应用程序。

11.3.1　准备工作

为了学习本节，你需要安装 Visual Studio 2012 和 Windows8（或以上）操作系统。除此之外无需其他准备。本节的源代码放置在 7644_Code\Chapter11\Recipe2 目录中。

11.3.2　实现方式

请执行以下步骤来了解如何在通常应用程序中使用 WinRT：

1. 启动 Visual Studio 2012。新建一个 C# 命令行应用程序项目。

2. 在 Visual Studio **解决方案预览**中右键点击新建的项目并选择**除移项目**…菜单选项。

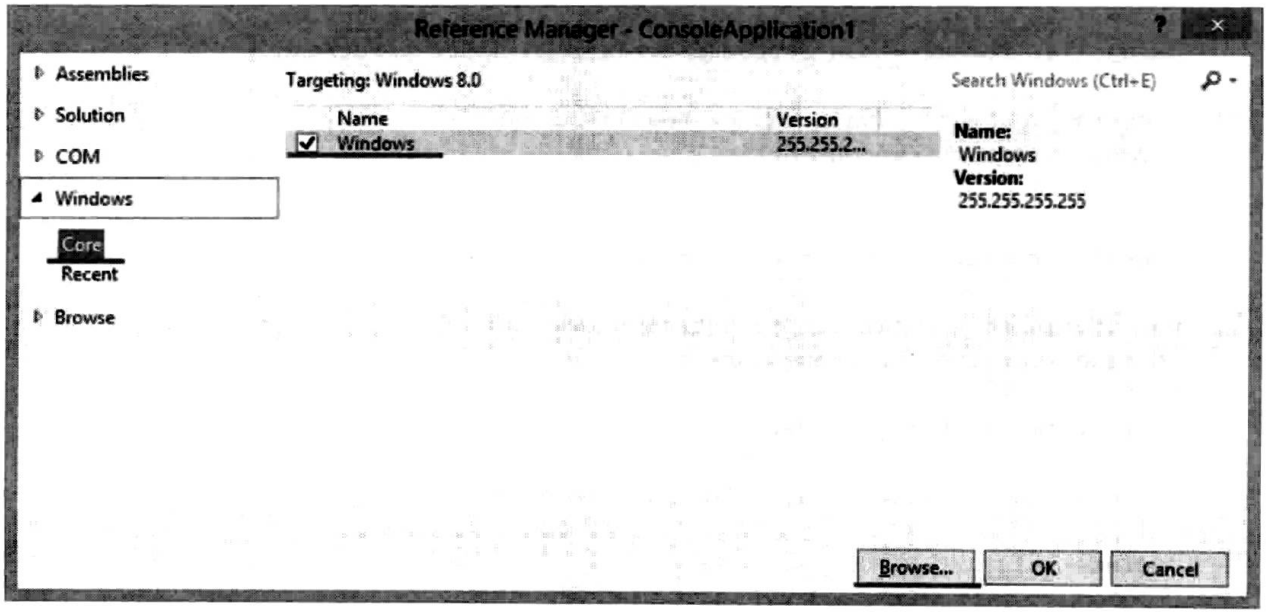
3. 右键点击被除移的项目并选择**编辑 ProjectName.csproj** 菜单选项。

4. 在 <TargetFrameworkVersion> 元素下添加如下 XML：

```
<TargetPlatformVersion>8.0</TargetPlatformVersion>
```

5. 保存 .csproj 文件，右键点击 Visual Studio **解决方案预览**中的被除移的项目，选择**重新加载项目**菜单选项。

6. 右键点击该项目并选择从 Windows 下的 Core 库中添加引用。然后点击**浏览**按钮。



7. 导航到 C:\Program Files\Reference Assemblies\Microsoft\Framework\.NETCore\v4.5 并点击 System.Runtime.WindowsRuntime.dll。

8. 在 Program.cs 文件中加入以下 using 指令：

```
using System;
using System.IO;
using System.Threading.Tasks;
using Windows.Storage;
```

9. 在 Main 方法下面加入以下代码片段：

```
async static Task AsynchronousProcessing()
{
  StorageFolder folder = KnownFolders.DocumentsLibrary;

  if (await folder.DoesFileExistAsync("test.txt"))
  {
    var fileToDelete = await folder.GetFileAsync(
      "test.txt");
    await fileToDelete.DeleteAsync(
      StorageDeleteOption.PermanentDelete);
  }

  var file = await folder.CreateFileAsync("test.txt",
    CreationCollisionOption.ReplaceExisting);
  using (var stream = await file.OpenAsync(FileAccessMode.
ReadWrite))
  using (var writer = new StreamWriter(stream.AsStreamForWrite()))
  {
    await writer.WriteLineAsync("Test content");
    await writer.FlushAsync();
  }

  using (var stream = await file.OpenAsync(FileAccessMode.Read))
  using (var reader = new StreamReader(stream.AsStreamForRead()))
  {
    string content = await reader.ReadToEndAsync();
    Console.WriteLine(content);
  }

  Console.WriteLine("Enumerating Folder Structure:");

  var itemsList = await folder.GetItemsAsync();
  foreach (var item in itemsList)
  {
    if (item is StorageFolder)
    {
      Console.WriteLine("{0} folder", item.Name);
    }
    else
    {
```

```
      Console.WriteLine(item.Name);
    }
  }
}
```

10. 在 Main 方法中加入以下代码片段：

```
var t = AsynchronousProcessing();
t.GetAwaiter().GetResult();
Console.WriteLine();
Console.WriteLine("Press ENTER to continue");
Console.ReadLine();
```

11. 在 Program 类定义下面加入以下代码片段：

```
static class Extensions
{
  public static async Task<bool> DoesFileExistAsync(this
    StorageFolder folder, string fileName)
  {
    try
    {
      await folder.GetFileAsync(fileName);
      return true;
    }
    catch (FileNotFoundException)
    {
      return false;
    }
  }
}
```

12. 运行程序。

11.3.3 工作原理

这里我们使用了一种很取巧的方式来在普通的 .NET 命令行程序中使用 WinRT API。遗憾的是，该场景并不适用所有的 API，但仍可以支持运动传感器、GPS 定位服务等。

为了在 Visual Studio 中引用 WinRT，通常需要手动编辑 .csproj 文件指定应用程序的目标平台为 Windows 8。然后手动引用 System.Runtime.WindowsRuntime.dll 来获取 WinRT 异步操作的 GetAwaiter 扩展方法实现。这可以让我们直接对 WinRT API 使用 await。这也是向后兼容的。当创建一个 WinRT 库时，我们不得不对异步操作暴露 WinRT 原生的 IAsyncOperation 接口族，从而可以被 JavaScript 和 C++ 以语言无关的方式使用。

WinRT 的文件操作是自描述的。我们可以使用异步创建文件和删除文件的操作。当然，WinRT 中的文件操作包括安全限制，鼓励你为你的应用程序使用特定的 Windows 文件夹，不允许你操作磁盘中的任意文件路径。

11.4 在 Windows 商店应用中使用 BackgroundTask

本节讲述了在 Windows 商店应用中创建后台任务的过程。该后台任务会更新桌面上该应用的动态卡片。

11.4.1 准备工作

为了学习本节，你需要安装 Visual Studio 2012 和 Windows8（或以上）操作系统。除此之外无需其他准备。本节的源代码放置在 7644_Code\Chapter11\Recipe3 目录中。

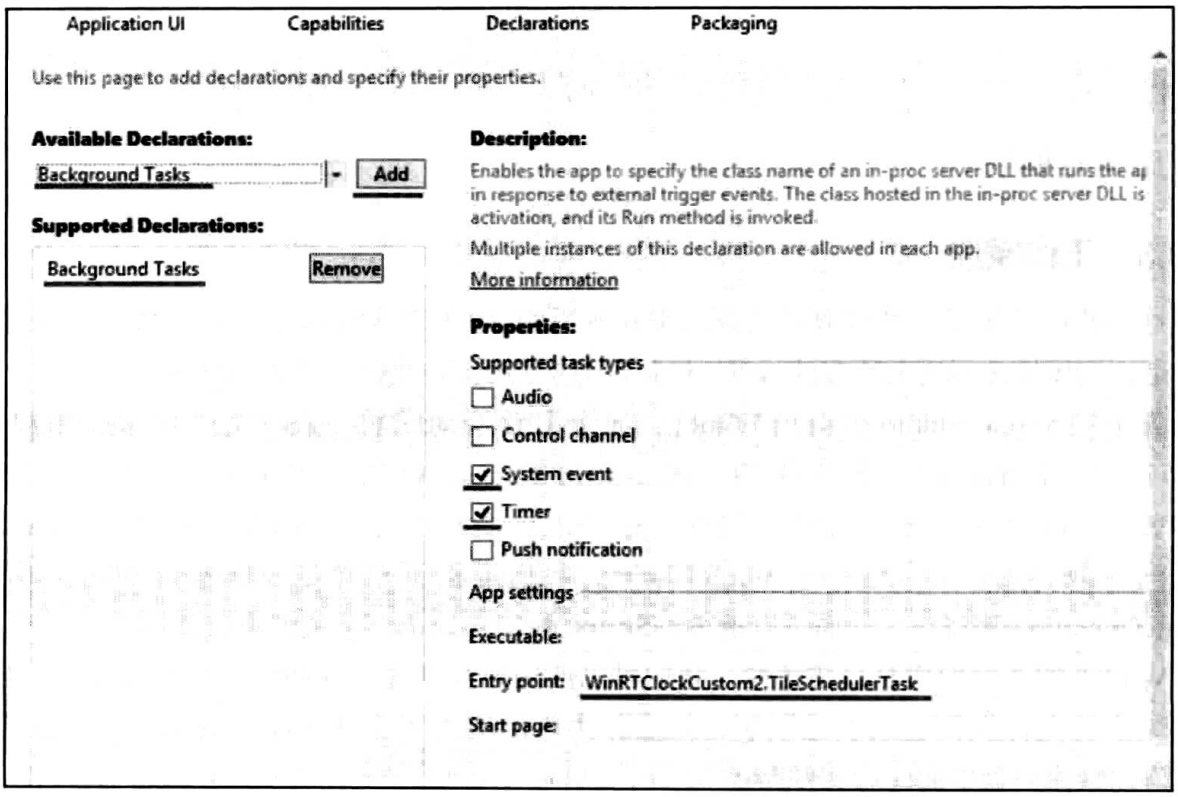
11.4.2 实现方式

请执行以下步骤来了解如何在 Windows 商店应用中使用 BackgroundTask：

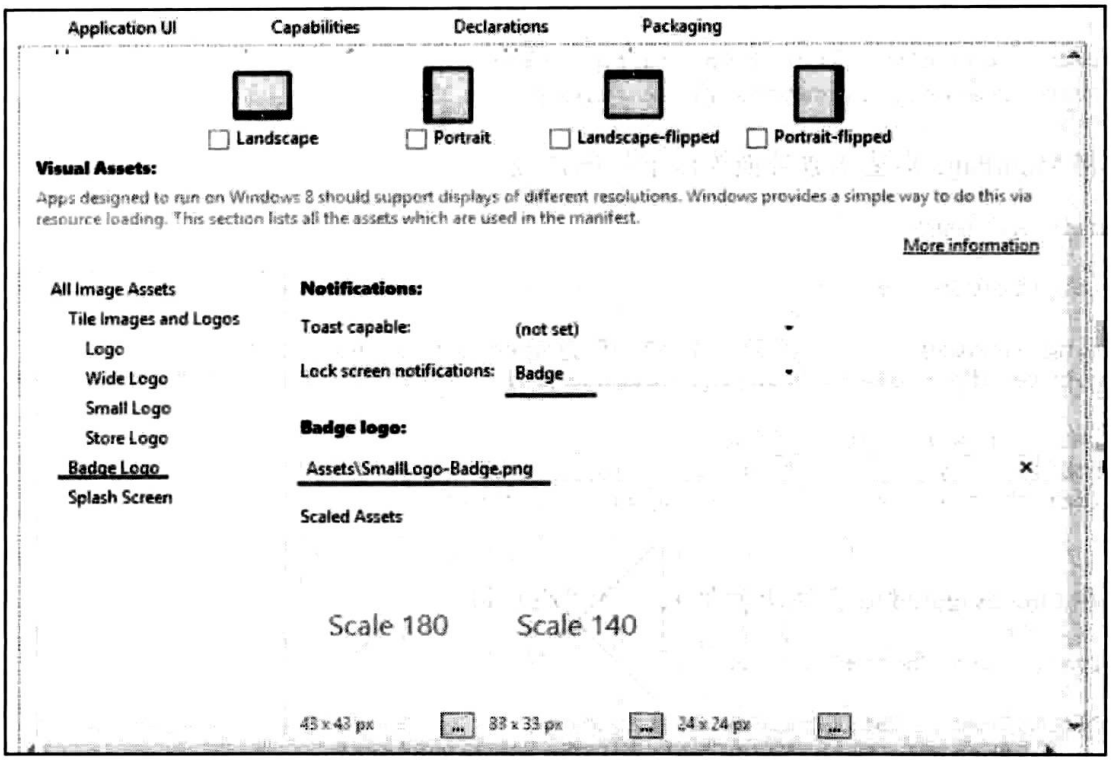
1. 启动 Visual Studio 2012。创建一个 Windows Store 类别下的 C# **空白应用**（XAML）项目。如果你需要更新开发人员许可证，请参见 11.2 节来获取详细的介绍。

2. 在**资源**目录中，在绘图编辑器中打开 SmallLogo.png 文件，剪贴为 24 × 24 像素大小，另存为资源目录中的 SmallLogo-Bage.png 文件，并引入到项目中。

3. 打开 Package.appxmanifest 文件。在**声明**标签中，为**支持的声明选项**添加**背景任务**。在属性中，点击支持的属性**系统事件**和**计时器**，并选择 Entry point 为 YourNamespace.TileSchedulerTask。YourNamespace 应该是你的应用的命名空间。



4. 在应用程序界面标签中，选择 Lock screen notifications as Badge，以及 Badge logo as Assets\SmallLogo-Badge.png。



5. 在 MainPage.xaml 文件中，在 Grid 元素中插入以下 XAML：

```
<StackPanel Margin="50">
  <TextBlock Name="Clock"
             Text="HH:mm"
             HorizontalAlignment="Center"
             VerticalAlignment="Center"
             Style="{StaticResource HeaderTextStyle}"/>
</StackPanel>
```

6. 在 MainPage.xaml 文件中加入以下 using 指令：

```
using System;
using System.Diagnostics;
using System.Globalization;
using System.Linq;
using System.Xml.Linq;
using Windows.ApplicationModel.Background;
using Windows.Data.Xml.Dom;
using Windows.System.UserProfile;
using Windows.UI.Notifications;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Navigation;
```

7. 在 MainPage 构造函数定义下面加入以下代码片段：

```
private const string TASK_NAME_USERPRESENT =
  "TileSchedulerTask_UserPresent";
private const string TASK_NAME_TIMER =
  "TileSchedulerTask_Timer";

private readonly CultureInfo _cultureInfo;
private readonly DispatcherTimer _timer;
```

8. 将 MainPage 构造函数替换为以下代码片段：

```
public MainPage()
{
InitializeComponent();

string language = GlobalizationPreferences.Languages.First();
_cultureInfo = new CultureInfo(language);

_timer = new DispatcherTimer();
_timer.Interval = TimeSpan.FromSeconds(1);
_timer.Tick += (sender, e) => UpdateClockText();
}
```

9. 在 OnNavigatedTo 方法上面加入一下代码片段：

```
private void UpdateClockText()
{
  Clock.Text = DateTime.Now.ToString(
    _cultureInfo.DateTimeFormat.FullDateTimePattern);
}

private static async void CreateClockTask()
{
  BackgroundAccessStatus result = await
    BackgroundExecutionManager.RequestAccessAsync();
  if (result == BackgroundAccessStatus.
    AllowedMayUseActiveRealTimeConnectivity ||
      result == BackgroundAccessStatus.
        AllowedWithAlwaysOnRealTimeConnectivity)
  {
    TileSchedulerTask.CreateSchedule();

    EnsureUserPresentTask();
    EnsureTimerTask();
  }
}

private static void EnsureUserPresentTask()
{
  foreach (var task in BackgroundTaskRegistration.AllTasks)
    if (task.Value.Name == TASK_NAME_USERPRESENT)
      return;

  var builder = new BackgroundTaskBuilder();
  builder.Name = TASK_NAME_USERPRESENT;
  builder.TaskEntryPoint =
    (typeof(TileSchedulerTask)).FullName;
```

```
  builder.SetTrigger(new SystemTrigger(
    SystemTriggerType.UserPresent, false));
  builder.Register();
}

private static void EnsureTimerTask()
{
  foreach (var task in BackgroundTaskRegistration.AllTasks)
    if (task.Value.Name == TASK_NAME_TIMER)
      return;

  var builder = new BackgroundTaskBuilder();
  builder.Name = TASK_NAME_TIMER;
  builder.TaskEntryPoint = (typeof(
    TileSchedulerTask)).FullName;
  builder.SetTrigger(new TimeTrigger(180, false));
  builder.Register();
}
```

10. 在 OnNavigatedTo 方法中加入以下代码片段：

```
_timer.Start();
UpdateClockText();
CreateClockTask();
```

11. 在 MainPage 类定义下面加入以下代码片段：

```
public sealed class TileSchedulerTask : IBackgroundTask
{
  public void Run(IBackgroundTaskInstance taskInstance)
  {
    var deferral = taskInstance.GetDeferral();
    CreateSchedule();
    deferral.Complete();
  }

  public static void CreateSchedule()
  {
    var tileUpdater = TileUpdateManager.
CreateTileUpdaterForApplication();
    var plannedUpdated = tileUpdater.
GetScheduledTileNotifications();

    DateTime now = DateTime.Now;
    DateTime planTill = now.AddHours(4);

    DateTime updateTime = new DateTime(now.Year, now.Month,
      now.Day, now.Hour, now.Minute, 0).AddMinutes(1);
    if (plannedUpdated.Count > 0)
      updateTime = plannedUpdated.Select(x =>
        x.DeliveryTime.DateTime).Union(new[] { updateTime
          }).Max();
    XmlDocument documentNow = GetTilenotificationXml(now);

    tileUpdater.Update(new TileNotification(documentNow) {
```

```
      ExpirationTime = now.AddMinutes(1) });

    for (var startPlanning = updateTime;
      startPlanning < planTill; startPlanning =
        startPlanning.AddMinutes(1))
    {
      Debug.WriteLine(startPlanning);
      Debug.WriteLine(planTill);

      try
      {
        XmlDocument document = GetTilenotificationXml(
          startPlanning);

        var scheduledNotification = new
          ScheduledTileNotification(document,
            new DateTimeOffset(startPlanning))
        {
          ExpirationTime = startPlanning.AddMinutes(1)
        };

        tileUpdater.AddToSchedule(scheduledNotification);
      }
      catch (Exception ex)
      {
        Debug.WriteLine("Error: " + ex.Message);
      }
    }
  }

  private static XmlDocument GetTilenotificationXml(
    DateTime dateTime)
  {
    string language =
      GlobalizationPreferences.Languages.First();
    var cultureInfo = new CultureInfo(language);

    string shortDate = dateTime.ToString(
      cultureInfo.DateTimeFormat.ShortTimePattern);
    string longDate = dateTime.ToString(
      cultureInfo.DateTimeFormat.LongDatePattern);

    var document = XElement.Parse(string.Format(@"<tile>
    <visual>
      <binding template=""TileSquareText02"">
        <text id=""1"">{0}</text>
        <text id=""2"">{1}</text>
      </binding>
      <binding template=""TileWideText01"">
        <text id=""1"">{0}</text>
        <text id=""2"">{1}</text>
        <text id=""3""></text>
        <text id=""4""></text>
      </binding>
    </visual>
  </tile>", shortDate, longDate));
```

```
    return document.ToXmlDocument();
  }
}

public static class DocumentExtensions
{
  public static XmlDocument ToXmlDocument(this
    XElement xDocument)
  {
    var xmlDocument = new XmlDocument();
    xmlDocument.LoadXml(xDocument.ToString());
    return xmlDocument;
  }
}
```

12. 运行程序。

11.4.3 工作原理

上面的程序展示了如何创建一个后台的计时器任务，以及如何通过该任务更新 Windows 8 起始屏幕上的动态卡片。编写 Windows 商店应用本身充满挑战，你不得不注意应用会挂起 / 恢复自身状态，以及其他很多事项。这里我们只关注我们的主要任务，而忽略其他问题。

我们的主要目标是当应用自身不处于前台时运行一些代码。首先创建一个 IBackgroundTask 接口的实现。当等到触发信号时 Run 方法会被调用。如果 Run 方法中含有 await 编写的异步代码，需要使用特殊的延迟对象（deferral object）来显式地指定何时开始和停止 Run 方法的执行，正如本章中所示。这点相当重要。在本例中，方法调用是同步的，但能说明使用延迟对象的需求。

我们在 Run 方法中的任务中创建了一组标题更新，将在 4 个小时中每分钟更新一次，并借助于 ScheduledTaskNotification 类将该任务注册在 TileUpdateManager 中。卡片使用了一个特殊的 XML 格式来指定如何具体显示卡片上的文本。当通过系统触发任务时，将安排一分钟一次的卡片更新，并持续 4 个小时。然后，我们需要注册该后台任务。它将被注册两次。第一次注册提供 UserPresent 触发器，这意味着当用户登录时会触发该任务。另一个触发器是一个时间触发器，其会每 3 小时运行一次该任务。

当程序运行时，创建了一个计时器，当应用处于前台时该计时器会运行。同时它会尝试注册后台任务。程序需要权限来注册任务，它会显示一个对话框来向用户要求权限。之后我们就安排了之后 4 个小时内的动态卡片的更新。如果关闭了应用，该活动卡片依然会在每分钟显示最新时间。三个小时后计时器会再运行一次后台任务，这将安排另一个活动卡片的更新。